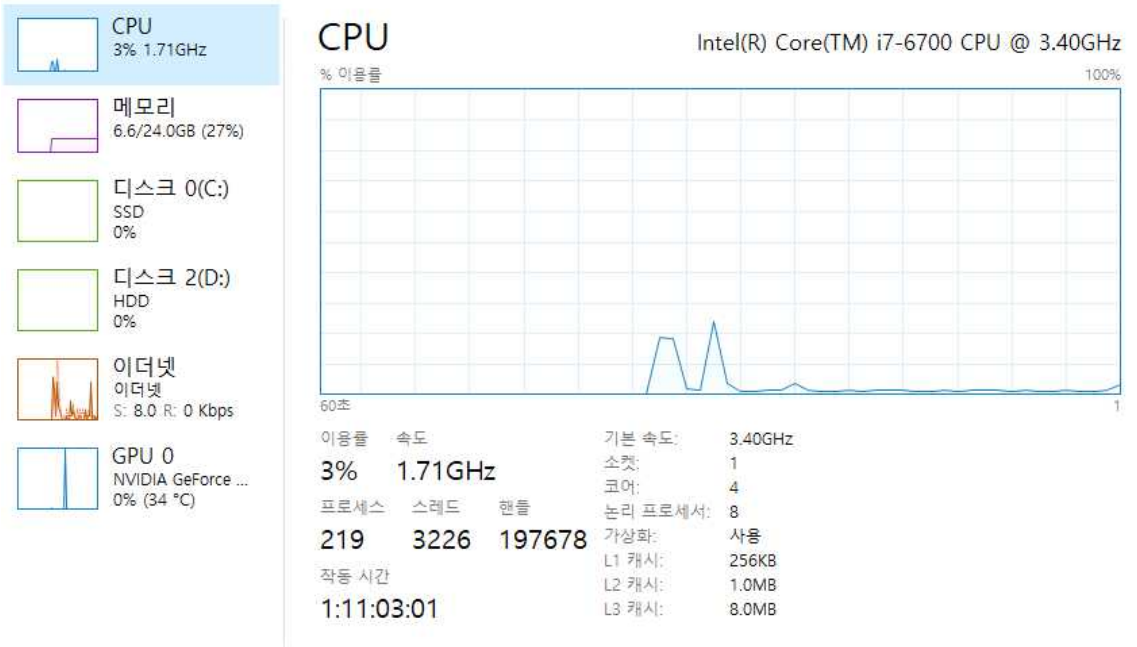
알고리즘 설계와 분석

20171300 지용환

과제 2

개발환경:



전반적인 컴퓨터 환경이다. window 10 운영체제이고, 모든 시간 비교는 visual studio의 원격 접속 기능을 이용해 putty (서버는 cspro10.sogang.ac.kr에서 주로 테스트) 에 로그인하고, 그 안에서 테스트 하며 시간을 비교했다.

알고리즘에 관해:

이번 과제에서 구현한 sort는 총 4가지로 각각 insertion sort, quick sort, merge sort, introsort이다.

먼저 insertion sort는 수업시간에 배웠듯이, 배열의 모든 것을 루프로 돌며, 한번 돌때마다, 그 인덱스 전의 것들을 검사함으로써 시간복잡도는 O(n^2)이 나온다. 중요한 점은 insertion sort는 이미 정렬된 순서에 대해서는 그냥 모든 루프를 돌며 한번씩만 검사를 하기에 O(n)이란 점이다. 이러한 특징 덕에, 본인이 구현한 intro sort에선 insertion sort를 활용하는데, 의미는 검사할 배열의 갯수가 적다면, 그냥 정렬된거나 다름없기에 이땐 insertion sort가 최고의 속도를 내기 때문이다. intro sort를 설명할 때 추가적으로 설명하겠다.

quick sort는 말 그대로 가장 빠른 sort 알고리즘으로 O(n)의 시간복잡도를 가진 partion 함수로 pivot의 위치를 정하고, pivot을 기준 좌우로 재귀적으로 quicksort를 호출하며 sort하는 방법이다. 시간복잡도는 O(nlogn)인데, quick이 가장 빠른 이유는 불필요한 데이터의 이동을 줄이며 데이터를 교환하고, 선정된 pivot은 이후 연산에서 제외되기 때문인데 실제로 random배열에서 quicksort는 가장 빠른 속도를 가진다. 하지만 quicksort또한 큰 단점이 있는데, 이 정렬 알고리즘은 pivot을 잘못 뽑으면 결과적으로 1~2개의 숫자만 자리를 찾음으로써, 모두다 정렬하려면 시간복잡도가 O(n^2)까지 올라간다는 점이다. 더욱이, 재귀적으로 호출하기에 pivot을 잘못뽑으면 너무 많은 재귀를 하다 그냥 overflow가 나서 sort가 안될 수도 있다. 즉, 최고의 시간복잡도를 가지는 sort 알고리즘을 구현하기 위해선 이 잘못된 pivot을 뽑게되는 경우를 방지해주는 quick sort를 만들어 주는 것인데, 이것이 4번의 intro sort이다.

merge sort는 일반적인 시간복잡도 O(nlogn)을 가진 sort 알고리즘이다. 장점은 어떠한 경우에도 그냥 nlogn의 시간복잡도를 가진다는 것인데, 단점은 정말 평범한 것이다. merge sort는 divide and conquer방식을 통해 하나가 될 때까지 계속 divde한 이후, base가 되면 비교 그리고 이후부터 비교를 하며 combine을 하는 방식인데, sort가 되있건 말건, 같은 횟수의 연산을 진행하며, 최종 단계가 되어서야 모든 자리가 선정되는 것이기에(매번 자리가 정해지는 quick과는 다름) quick보다는 느리다. 물론 아주 느리진 않지만 가장 빠른 알고리즘과는 거리가 먼 알고리즘이다.

introsort는 말했듯이 quick sort의 단점을 줄이는데에 초점을 둔 sort이다. 하지만 그전에, insertion sort는 비교를 하는 배열의 사이즈가 적으면 최상의 시간복잡도를 가진다고 설명을 했다. 이유는 컴퓨터 상에서 5개의 데이터가 있을때 그냥 5개를 sort하는 속도나, 그냥 5개를 그냥 읽는 속도(O(n))나 유사하기 때문이다. 실험을 한결과 약 사이즈가 16까지는 insertion이 최상의 속도를 보여주었다.(17부터 비슷비슷하다 슬슬 느려진다) 즉, introsort는 quicksort를 하다가, size가 16이하가 되면 그냥 insertion으로 진행시켜 그냥 quick만 진행하는 것보다는 더 빠른 속도를 만들어준다. insertion sort를 추가한 것보다 중요한 것은 이미 정렬된 배열을 정렬할때의 quick의 단점을 없애는 것인데, 이방법은 median of 3방법으로 없앨수 있다. median of 3는 처음, 중간, 끝인덱스 중 중간값을 pivot으로 만들어 버려서 그 pivot의 위치를 찾게 하는 것인데, 이렇게 되면 이미 정렬된 배열에서 가장 빠른 속도를 보여준다. 이유는 안그래도 빠른 quick이 median of 3방법을 쓸시, 정확히 반씩 분할을 하며 훨씬더 적은 연산으로 sort가 가능하기 때문이다. 하지만 이또

한 문제가 있는데 배열이 killer of median of 3로 주어졌을 경우이다. 간단히, median of 3의 값이 배열의 최댓값과 같거나 살짝 작은 경우로 생각하면 된다. 이렇게 되면 median of 3를 쓴다한들 결국에 1~3개를 제외한 나머지 배열에 대해 다시 quick을 부르게 되고, 다시말해 분할자체가 엉망이 되어 이미 정렬된 배열에대해 quick sort를 쓰는 시간복잡도와 별반 다를 것이 없어지게 된다. introsort의 최종 목표는 이 경우를 막아주는 것인데, 간단히 최대 연산 횟수를 지정해주어서 그 횟수가 넘어간다면 그냥 quick을 포기하고 merge를 해버리는 것이다. 주로 이 횟수는 2*log(사이즈)로 설정이 된다. 이렇게 되면 당연히 quick의 분할을 아주 못하다가 포기하고 merge를 하는 것이기에 그냥 merge보다도 느리지만, quick보다는 빠르기에 median of 3를 그나마 막을수 있는 sort방법이다. 즉 ,시간 복잡도가 어떠한 경우에도 O(nlogn)이지만, quick보다는 대부분 빠르고 merge보다 빠를 가능성이 높은 sort이다.

실험 테스트(시간):
정렬된 오름차순 배열 1024개 : 단순히 0,2,4,6,8,10,12,,,식으로 증가하는 input이다. 마지막은 2046
insertion: 0.000013
quick sort: 0.005681
merge sort: 0.000249
intro sort: 0.000173
예상 했던 결과이다. 시간복잡도 O(n)이 된 insertion 이 가장 빠르고 median of three방법을 써 정렬된 배열의 sort단점을 벗어난 그리고 insertion의 장점을 가진 intro가 2등 이후 merge, 그리고 O(n^2)이된 quick sort의 순서로 빨랐다.

정렬배열 오름차순 100000개 : 0,1,....99999
insertion: 0.000561
quick: 23.402837
merge: 0.018164
intro: 0.007107
마찬가지의 순위가 나왔다. (insertion>intro>merge>quick)

정렬배열 내림차순 100000개 : 99999부터 0까지 감소하는 내림차순 배열이다.
insertion: 16.992845

quick: 19.647865
merge: 0.018225
intro: 0.021427
(merge>intro>insertion>quick)
오름차순의 결과와는 다르다. insertion은 오름차순 정렬일때만 n번 연산을 하지, 내림차순이면 정말 정직하게 n(n+1)/2번 연산을 하게 되어 시간이 높고, quick도 결국 pivot이 제대로 분할을 하지 못해 느리게 수행이 된다. 이때, 재귀적으로 계속 호출하는 quick이 더 느린 것을 확인 가능하다.(실제로 size가 조금더 크면 insertion은 시간이 걸려도 되긴되지만, quick은 그냥 안될 수도 있다.) merge는 평소대로 연산 하여 오름차순 시간과 유사했고, intro는 오름차순보단 느렸다. 이유는 intro에 사용되는 insertion sort이기 때문인데, 내림차순에서 insertion은 말그대로 최악이고, size가 작더라도, 결국에 16개를 오름차순하는 것보단 느리기 때문이다. 이것이 쌓이다 보니 merge보다 살짝 느린 결과를 보이게 되었다.

랜덤 배열 1024개 (4개의 평균): 되게 숫자의 범위를 크게 설정했지만, 컴퓨터 시스템상 40000이상의 숫자는 없었다. 하지만 정렬 시간측정에는 큰 의미가 없을 것으로 판단했고, 그냥 진행했다. 즉, 0~39999사이의 랜덤 숫자.
insertion: 0.002036
quick: 0.000261
merge: 0.000418
intro: 0.000233
예상대로 quick에서 조금더 빠르게 insertion을 추가시킨 intro가 1등, 이후 quick, merg, 그리고 그냥 O(n^2)인 insertion이 가장 느렸다.

랜덤배열 1000000개 (4개의 평균) : 마찬가지로 40000이상의 숫자는 본적 없는 것 같다. 0~39999의 랜덤 숫자
insertion: 너무 오래걸려 측정포기
quick: 0.239305
merge: 0.288555
intro: 0.229855
마찬가지의 순서가 나왔다.

median of 3 killer 1000개 : 정확히 만든 배열인지 확신할수 없지만 median

of 3의 값이 그냥 중간값이 항상 최댓값과 유사하게 만들어 주었다. 즉, 10개인 경우
-1 1 3 5 7 0 2 4 6 8처럼, 중간의값 (7)이 최댓값(8)과 유사, partition 이후엔 중간값(5)가 최댓값(6)과 유사 하여 median of 3를 써도 partition이 잘 안되는 경우이다. 1000개는 위의 10개의 예시를 그냥 확장한 것이다.
insertion: 0.000987
quick: 0.002779
merge: 0.000246
intro: 0.000411
위에서 설명한 결과와 동일하다. 즉, merge가 가장빠르고, median of three의 단점에서 그나마 벗어난 intro, 그이후 insertion이다. 여기서 quick보다 insertino이 빠른 이유는, 위의 예시로 든 배열에서 볼수있듯이 killer로 설정한 배열이 꽤나 많이 정렬되어있기 때문이다.(반을 기준으로 양쪽이 완전히 정렬)

더 많은 test결과는 다음 표로 나타내겠다.

배열 종류	데이터 갯수	insertion	quick	merge	intro
정렬 (오름차순)	10	0.000002	0.000004	0.000006	0.000021
	100	0.000002	0.000062	0.000038	0.000027
	1024	0.000013	0.005681	0.000249	0.000173
	10000	0.000104	0.277746	0.002840	0.001070
	100000	0.000561	23.402837	0.018164	0.007107
정렬 (내림차순)	10	0.000004	0.000003	0.000007	0.000021
	100	0.000042	0.000052	0.000027	0.000030
	1000	0.003822	0.004508	0.000251	0.000229
	10000	0.204478	0.228792	0.002792	0.003138
	100000	16.992845	19.647865	0.018225	0.021426
랜덤 (4개 평균)	10	0.000003	0.000004	0.000006	0.000019
	100	0.000024	0.000020	0.000038	0.000035
	1024	0.002036	0.000261	0.000418	0.000233
	10000	0.107724	0.003005	0.004778	0.002718
	100000	8.584699	0.020894	0.029504	0.018065
	1000000	포기	0.239305	0.288555	0.229855
median of 3 killer	1000	0.000987	0.002779	0.000246	0.000411

comment:
이번 과제로 가장크게 깨달은 점은 모든 경우에 완벽한 sort 알고리즘은 없는

것이다. 물론 있다면 그것만 배웠겠지만 말이다. c++언어에서 기본적으로 introsort를 지원해준다는 것을 본적이 있고, 예상한 결과는 사실 내림차순과 median of three killer를 제외한 모든 경우에 intro가 최상의 결과를 보여줄 것이라 예상했다. 하지만, 오름차순에서 최고가 아닐뿐더러, 그냥 input갯수가 적을땐 introsort는 최악이었다. 오름차순같은 경우엔 재귀적으로 호출하며 걸리는 시간 때문에 그냥 insertion 하는것보단 느릴수도 있다고 납득은 했지만, input갯수가 가장 적은 10개 일땐 디버깅을 해도 단순히 insertion sort한번하고 끝난걸 확인했고, 결과과 insertion과 6배정도 느린이유는 사실 아직도 정확하겐 모르겠다. 예상되는 이유는 그냥 함수를 거치는 시간이 긴건가 하는 생각이 든다. introsort를 하기 위해선, depth를 구해야 하고, 그 depth를 구해주는 함수를 거쳐(depth한번 구한다고 시간이 저렇게 많이 걸릴 것 같지는 않다..,.) inrtosort에 본격적으로 돌입하는 그 시간이, 즉 2개의 함수를 거치는 시간이 약 0.000015초인 것 같고 이 함수를 거쳐가는 시간 때문에 input갯수가 100개까지는 intro가 그렇게 좋지는 않은 것을 깨달았다. 즉 시간을 줄이기 위해선 함수를 하나도 거치지 않고 그냥 main함수내에 sort 알고리즘을 짜면 좋겠지만,, 재귀적으로 호출을 해야하는 함수같은 경우엔 이것이 불가능 하기에 작은 input같은 경우엔 어짜피 시간이 적게 걸리니 크게 신경쓰지 않고 c++에서도 intro를 제공한 것 같다. 물론 세상엔 큰 input sort가 더 많겠지만, 작은 100개 이하의 input또한 엄청나게 많기에, 상황에 따라 시간복잡도만 생각하는 것이 아니라, 함수를 거치지 않고도 선언이 가능한지를 판단하며 가장 최선의 알고리즘을 구현해야 겠다는 것을 깨달았다.