CSE3081 (2 반) 알고리즘 설계와 분석

<프로그래밍 숙제 3>

Sorting Algorithm 구현과 시간 복잡도 분석을 위한 실험

20140938 임다은

1. 실험 환경

OS: Windows 10 Pro

CPU: 3.30GHz Intel Core i5-6600

RAM: 4GB (3.90GB 사용가능).

Compiler: Visual Studio Community 2015

2. 알고리즘 구현 관련 특이사항

1) 효율적인 구현을 위한 노력

- 생산성 높은 코드 구현을 위해 구조체를 서로 바꾸는 기능, 구조체를 채우는 기능 등은 함수로 대체하였다. 각 파일에 들어있는 swap, substitute function 이 그 예이다.
- Quicksort optimization 함수를 구현을 위한 최적화 기법
    i. 포인터를 최대한 적게 활용하기 위해 자주 접근하는 값은 따로 변수를 할당해서 미리 값을 받아놓고 사용했으며, 생산성을 높이기 위해 참조하는 변수를 만들어 놓은 것을 optimze 를 위해서는 다시 풀어서 썼다.
    ii. Negative value 를 절대 가지지 않는 변수들을 int 대신 unsigned int 로 치환
    iii. Pseudo 알고리즘 상에서 중복되는 코드 제외
- 함수를 구현하면서 동적 할당된 data 를 최대한 free 시켜주기 위해 노력했다.

2) 정확한 구현을 위한 노력

- 정확한 기능인지 확인 위한 알고리즘

Heap sort 구현 시 checkIfHeap 함수를 만들어 adjust 함수 호출 이후 제대로 된 heap 이 만들어지는 지를 확인하였다. Iteration 을 통해서 부모가 자식보다 더 큰지에 대해서 확인하는 함수이다.

3. 실험 방법

1) 실험 데이터 생성

총 세가지 실험 데이터가 필요했다. Entirely_random, Descending, Few_swaps 의 경우였는데, entirely_random 은 교수님이 만들어주신 파일을 이용하였고, Descending, Few_swaps 는 그 파일을 수정하여 만들었다. 데이터의 크기는 아주 작은 2^5 부터 2^2 배수로 커져 2^20 까지 만들었다. 이 때 각 파일의 이름은 안내된대로 (문제의 크기 n)_(random 또는 descending 또는 swaps) 형태이다.

2) 실험 과정

전체적인 실험은 2^10 부터 2^20 데이터까지 6 가지 데이터를 5 번 반복하여 진행하였다. 비쥬얼 스튜디오를 이용하여 상기 명시환 윈도우 환경에서 실험하였다. insertion sort 는 큰 데이터를 처리할 능력이 부족하여 작은 사이즈의 데이터에서 실험을 진행하였다. 실험 데이터는 sorting_time_experiment.txt 파일에 따로 걸린 시간 데이터를 저장하고, 엑셀을 이용하여 분석하였다.

4. 실험 측정 데이터

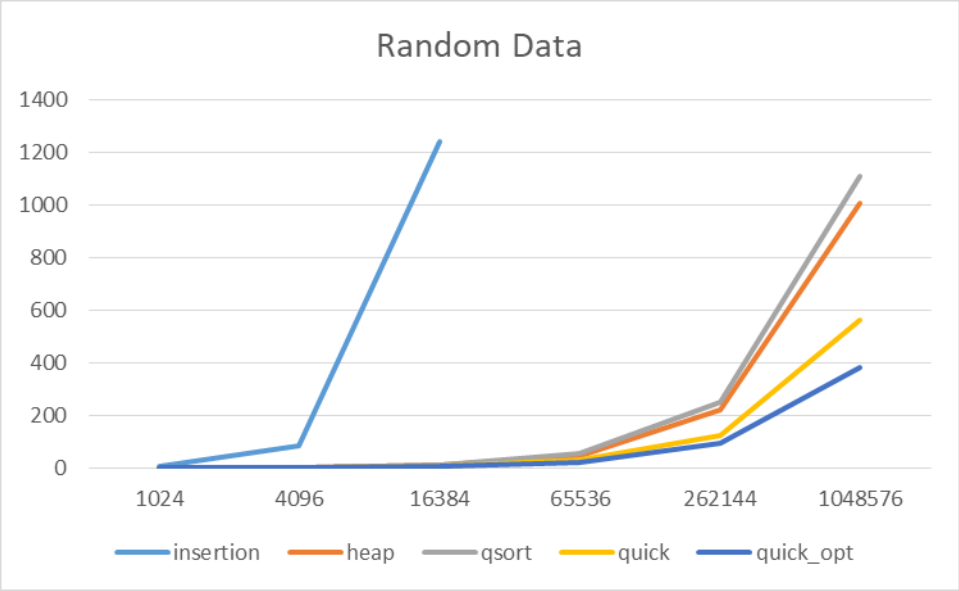
1) Raw Data

모든 측정 결과에 대한 raw data 표는 <첨부자료>에 별첨하였다.

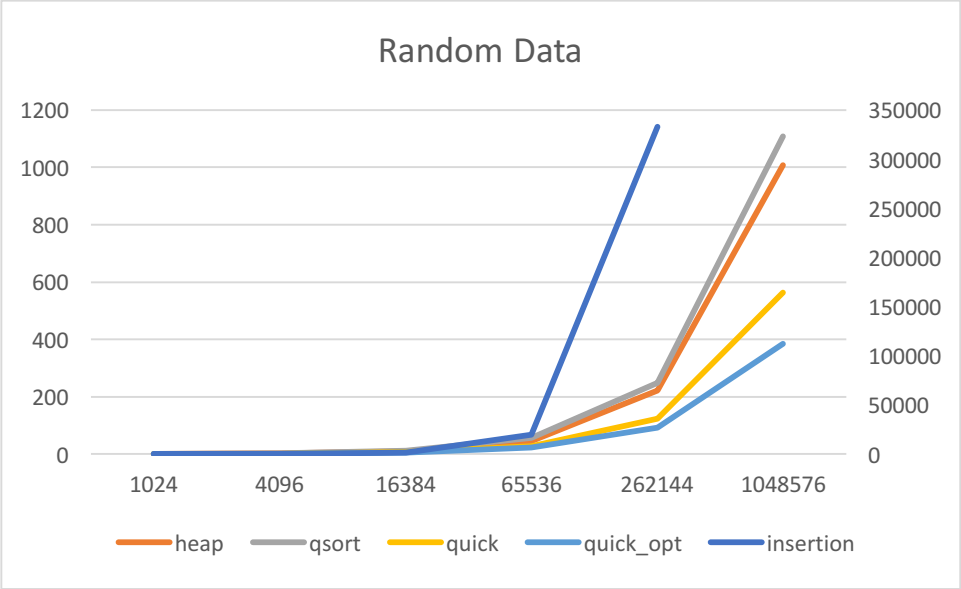
2) Entirely Random Data

RANDOM DATA					
Data Size	*insertion*	*heap*	*qsort*	*quick*	*quick_opt*
1024	6.2254	0.4826	0.8696	0.4028	0.4184
4096	82.6068	2.2756	3.0094	1.4418	1.0408
16384	1242.509	8.7126	12.3228	6.5048	5.1402
65536	19996.2046	44.536	55.917	27.829	22.1854
262144	332762.5374	221.9266	248.2042	124.2162	93.017
1048576		1007.196	1108.4158	562.4926	383.634

<표 1> Entirely Random Data 수행시간



<그림 1> Entirely Random Data 수행시간

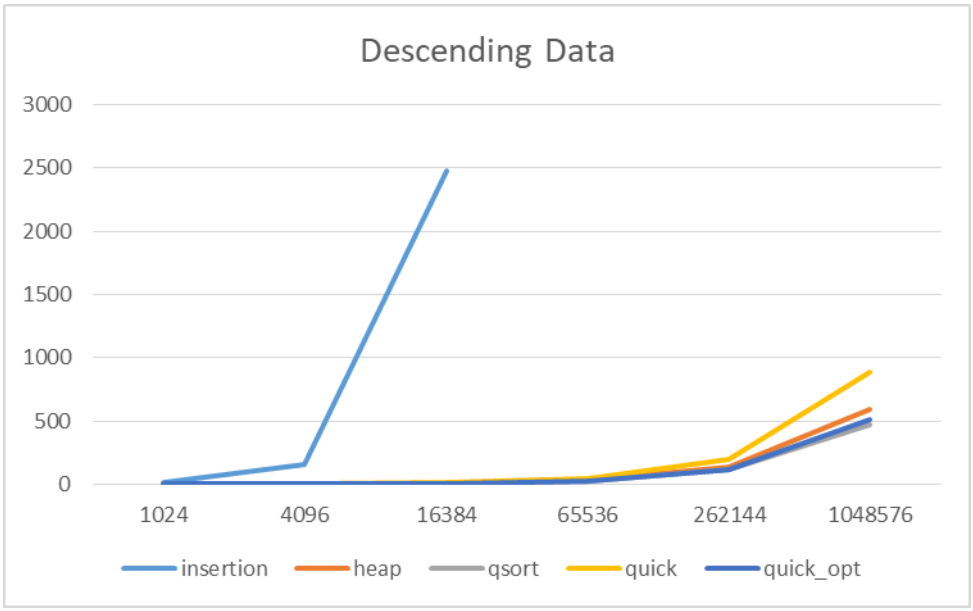


<그림 2> Random Data, inesrtion sort 는 오른쪽 축, 나머지 알고리즘은 왼쪽 축 중심

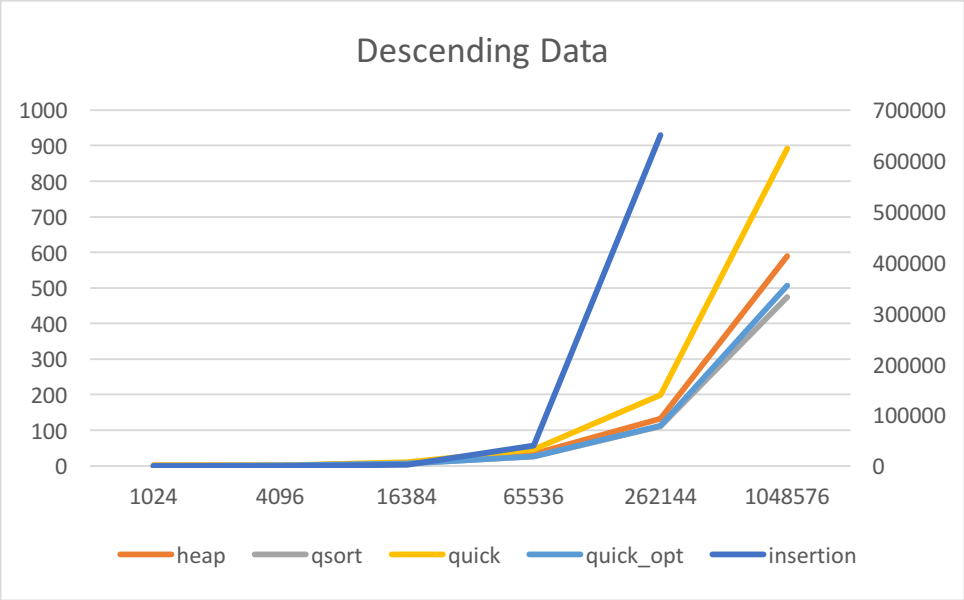
3) Descending Data

DESCENDING					
Data Size	*insertion*	*heap*	*qsort*	*quick*	*quick_opt*
1024	9.6608	0.379	0.436	0.5284	0.3254
4096	153.2552	1.7748	1.3746	2.1106	1.5178
16384	2482.2784	7.0834	5.9126	9.6784	5.8978
65536	40266.8124	33.7682	25.6302	45.3304	25.5984
262144	650517.7884	132.9704	111.1488	199.4738	112.0244
1048576		588.698	474.6986	890.8646	507.0142

<표 2> Descending Data 알고리즘 별 수행시간



<그림 3> Descending Data 알고리즘 별 수행시간

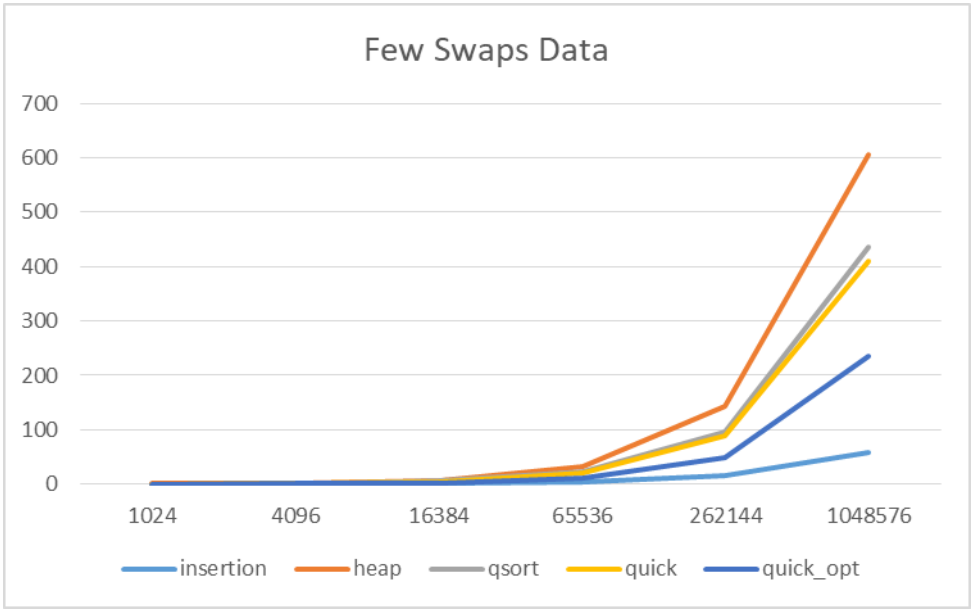


<그림 4>Decsending Data, inesrtion sort 는 오른쪽 축, 나머지 알고리즘은 왼쪽 축 중심

4) Few Swaps Data

FEW SWAPS					
Data Size	*insertion*	*heap*	*qsort*	*quick*	*quick_opt*
1024	0.107	0.6144	0.4092	0.2884	0.2046
4096	0.0606	0.4818	0.0874	1.6782	1.6148
16384	1.3806	7.4174	5.3546	4.8504	2.3712
65536	4.0694	32.348	22.85	20.295	10.5878
262144	15.0254	142.719	96.989	89.5752	48.313
1048576	58.3806	604.8094	436.1994	409.4664	234.7752

<표 3> Few Swaps Data 알고리즘 별 수행시간



<그림 5> Few Swaps Data 알고리즘 별 수행시간

5. 측정 결과 분석

1) 이론적 시간 복잡도와 실제 시간 복잡도 비교

data size 1024 기준				
Sorting Method	이론적 시간복잡도	수행시간		
		Random	Descending	Few Swaps
Insertion Sort	O(n^2)	6.2254	9.6608	0.107
Heap Sort	O(nlogn)	0.4826	0.379	0.6144
Qsort	O(nlogn)	0.8696	0.436	0.4092
Quick Sort	O(nlogn)	0.4028	0.5284	0.2884

<표 5> 이론적 시간복잡도와 실제 시간복잡도

O(n^2)의 시간복잡도를 갖는 insertion sort 와 O(nlogn)의 시간복잡도를 갖는 다른 알고리즘들 사이에 이를 미루어볼 때 이론적인 시간복잡도와 실제 수행시간 간에 밀접한 관계를 가지고 있음을 알 수 있다.

2) 서로 다른 데이터 형태에 따른 수행 시간차이

- Entirely Random Data 와 Descending Data 의 그래프에는 insertion sort 의 수행시간이 너무 길어짐으로 인해 정상적인 비교가 불가능하여, insertion sort 는 처음 3 개의 데이터만 실었다.
- Descending Data 에서도 마찬가지로 insertion sort 가 확연한 차이로 수행시간이 길었고, 나머지 알고리즘은 비슷한 수행시간을 보여주었다.
- Few Swap Data 에서 insertion sort 의 다른 sorting algorithm 과 확연한 차이를 보여준다. 즉 어느 정도 정렬된 data 에 한해서는 insertion sort 가 그 어떤 sorting algorithm 보다 우수한 성능을 보여준다는 것이 실험적으로 증명된 것이다.
- heap sort 는 random data 보다 Descending data 에서 더 좋은 성능을 보였으며, quick sort 는 random data 에서 더 좋은 성능을 보였다. 이는 heap sort 는 descending 하게 정렬된 경우 상대적으로 max heap 을 만들기 쉬워, 그에 대한 비용이 줄어든 것으로 생각되며, quick sort 와 같은 경우는 random 하게 섞여있을수록 pivot data 가 중간값에 가까워기 때문에 이와 같은 결과가 나온 것으로 해석된다.

3) Quick sort 의 시간 복잡도 분석

재귀적인 방법으로 직접 구현한 QUICK_SORT 와 QUICK_SORT_OPT 의 경우 Random Data 에서는 확연하게 heapsort 와 insertion sort 보다 우수한 성능을 보여주었다. 또한 descending data 에서는 optimize 된 quick sort 와 qsort 가 우수한 수행속도를 보여주었다. Few data swap 의 경우에도 data 정렬 특성 상 insertion sort 가 for 문을 수행하는 일이 거의 없어 정렬 속도가 빠른 것을 제외하면 좋은 성능을 보여주었다. 이를 미루어 볼 때 quick sort 가 대체적으로 다른 sorting algorithm 에 비해 정렬 속도가 빠름을 알 수 있었다.

4) Insertion sort 의 시간 복잡도 분석

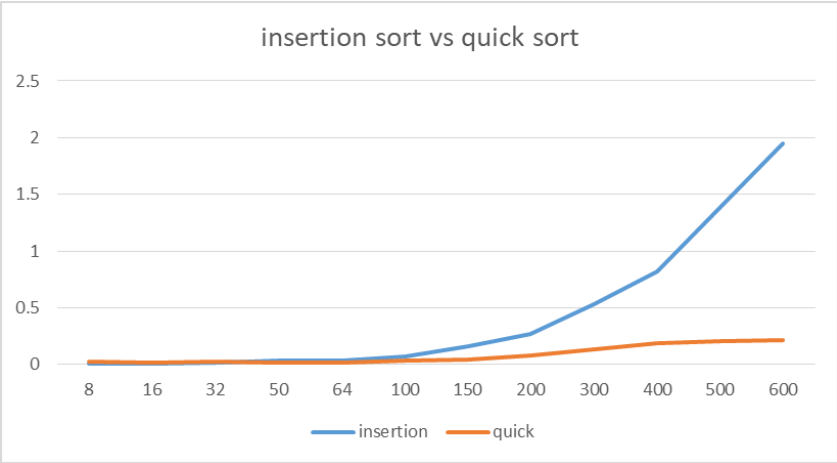
<표 3>과 <그림 5>를 보면 Few Swaps Data 의 실험결과에서 insertion sort 가 가장 우수한 성능을 보여줌을 알 수 있다. for 문을 돌면서 현재 검사하는 데이터의 맞는 위치를 찾아주는 insertion sort 의 특성 상 어느 정도 정렬된 데이터라면 자신의 위치를 찾기 위한 이동이 별로 필요하지 않을 것이다. 따라서 다른 sorting algorithm 에 비해 더 빠르게 정렬될 수 있는 것이다. 완벽하게 정렬되어있는 데이터라면 반복문을 돌 때 데이터의 이동이 아예 없을 것이므로 n 개의 데이터의 자리가 맞는지 한 번만 검사하게 되어 O(n)만큼의 시간복잡도를 가질 것이다.

5) Quick sort 와 Insertion sort 시간 복잡도 비교

Insertion 과 quick sort 가 작은 사이즈에서 완전히 random 한 data 를 정렬하는 시간을 비교하기 위한 실험을 진행하였다.

size	*insertion*	*quick*	*ins-quick*	*ins/quick*
8	0.0068	0.0244	-0.0176	0.278688525
16	0.0096	0.0146	-0.005	0.657534247
32	0.0118	0.0236	-0.0118	0.5
50	0.0346	0.0118	0.0228	2.93220339
64	0.0314	0.0154	0.016	2.038961039
100	0.069	0.0312	0.0378	2.211538462
150	0.1556	0.0466	0.109	3.339055794
200	0.2642	0.0768	0.1874	3.440104167
300	0.5316	0.1304	0.4012	4.076687117
400	0.8218	0.1838	0.638	4.471164309
500	1.3922	0.2046	1.1876	6.804496579
600	1.9474	0.2096	1.7378	9.291030534

<표 6> insertion 과 quick 정렬의 데이터 사이즈 별 평균 수행시간



<그림 6> insertion 과 quick 정렬의 데이터 사이즈 별 평균 수행시간

표의 데이터를 보면 insertion sort 와 quick sort 의 차이는 거의 없거나, 유의미한 경향성을 보이지 않다가 data size 가 100 을 넘어서게 되면 수행 시간 차이와 비율에서 급격하게 차이가 생김을 알 수 있다. 이를 그래프로 표현하면 그림 1 과 같이 사이즈 100 부터 확연한 차이를 내며 벌어지는 것을 알 수 있다. 따라서 quick sort 와 insertion sort 를 사용해도 크게 문제가되지 않는 충분히 작은 n 값은 100 이라고 결론 내릴 수 있다.

6) Quick sort 의 트리의 깊이에 대한 실험적 분석

Size N	Tree depth	logN	N
1024	588	10	1024
4096	2345	12	4096
16384	9340	14	16384
65536	37490	16	65536
262144	149824	18	262144
1048576	600817	20	1048576

<표 7> Quick sort tree depth

표 7 은 quick sort 를 수행했을 때 tree 의 depth 이다. 최악의 경우인 O(N)까지 depth 가 올라가진 않지만, median 값을 기준으로 정확하게 partition 되었을 때의 값인 logN 보다는 훨씬 큰 만큼의 depth 가 나왔다. 즉 어느정도 랜덤하게 pivot 을 정해주면 최악의 tree depth 를 갖게되는 것을 방지할 수 있다는 것이다.

6. 과제 수행 후 발전 사항

1) 구현 시 겪은 어려움

- array 의 인덱스 조정 및 변수의 정확한 의미 선정

각 함수를 구현하는 데 가장 많은 디버깅이 필요했던 부분은 조금씩 인덱스가 어긋나는 것이었다. 예를 들어 selection recursion 에서 k 번째 숫자를 찾는 것에 대해서 함수에서는 int k 라는 변수의 의미를 'k 번째 수가 있는 인덱스다' 라고 명시적으로 이야기 한 후에 그에 따른 recursion 을 만들어줄 때 제대로 함수가 동작할 수 있었다. 상술하자면, 이 함수에서 int s1, s2, s3 는 같은 이름을 갖는 집합에 속하는 원소의 개수를 명시하는데, data 배열의 인덱스를 조정하고, 다시 recursion 할 때는 어떤 인덱스에서 어떤 수를 찾고자 할 때 주의가 필요했다.

- 구조체와 포인터를 사용하는 것에서 어려움을 겪었다.

insertion sort 에서 tmp 변수에 잠시 검사하고자 하는 data 를 담아두고자 할 때 단순히 포인터로 이 데이터를 가리키게만 만들면 substitute(&tmp, &data[cur])와같이 함수로 구조체를 넘겨서 안의 데이터를 옮기려고 시도할 때 오류가 발생했다. 따라서 tmp 변수를

포인터로 구조체를 가리키게 만들면 안되고, 구조체 자체로 선언하여 구조체의 내용을 모두 담아놓는 것이 필요했다.

2) 경험적으로 알게된 사항

- compiler 마다 오류를 잡아내는 방법이 달랐다.

로컬인 mac 에서는 돌아가는 코드가 linux 에서 컴파일 할 때 segmentation fault 가 발생했었다. 오류를 찾아내기 위해 gdb 를 통해서 디버깅을 해 본 결과 문제는 조건문 체크시 잘못된 메모리 참조를 하는 것에 있었다.

```
for( i = left+1; i <= right ; i++){
    cur = i;
    substitute(&tmp, &data[cur]);
    curkey = ELEMENT_KEY(&tmp);
    while(cur > left && curkey < ELEMENT_KEY(&data[cur-1])){
        /***************8fatal ERROR!!!!**********************/
        // if cur becomes 0 because of c--; below,
        // in the condition check sentence data[-1] is accessed!!!!
        // if cur > left should be checked first!!
        substitute(&data[cur], &data[cur-1]);
        cur--;
    }
    substitute(&data[cur], &tmp);
}
```

<그림 7> insertion sorting code

원래 while 문의 조건문은 'curkey < ELEMENT_KEY(&data[cur-1]) && cur > left' 였다. cur 이 -1 이 되면 cur > left 조건에 배치되어 while 문을빠져나와야 한다. 그러나 주석에 명시한 것과 같이 c--가 일어났을 때 데이터 값에 대한 확인을 먼저하게되면 잘못된 메모리를 먼저 참조하게 되므로 오류가 발생하기때문에 segmentation fault 가 발생할 수 있다. 따라서 cur > left 조건을 먼저 체크해야 한다.

- makefile 만드는 것

로컬 컴퓨터로 맥을 사용하여서 기본적인 구현은 터미널 상에서 하고, gcc 를 이용하여 컴파일 했다. 이를 위해 makefile 을 만들었는데, 같이 컴파일 되는 파일에 같은 이름의 함수가있으면 링크 오류가 발생한다.

또한 makefile 을 고치고 난 이후에는 make clean 을 한 이후에 make 커맨드를 실행해야 한다.

- 포인터 연산

selection recursion 에서 메모리 포인터를 연산할 때 s1, s2 는 구조체의 수를 나타내는 int 타입이다. memcpy(S3, data+s1+s2, sizeof(ELEMENT)*(elemCnt - s1-s2))와 같이

data 배열에 단순 덧셈연산을 하는 것이 memcpy 연산에서도 sizeof 함수를 쓰지 않고 가능함을 알 수 있었다.

<첨부자료>

측정결과분석-1)

INSERTION							
Data Format	Data Size	1	2	3	4	5	average
random	1024	5.317	8.955	5.721	6.085	5.049	6.2254
	4096	81.377	84.88	82.878	82.194	81.705	82.6068
	16384	1225.286	1240.758	1247.203	1244.773	1254.525	1242.509
	65536	19593.637	19695.203	20779.24	19922.781	19990.162	19996.2046
	262144	348387.656	319822.063	329523.719	338257.236	327822.013	332762.537
	1048576	?	?	?	?	?	
descending	1024	9.605	9.949	9.763	9.431	9.556	9.6608
	4096	151.957	159.95	149.815	154.001	150.553	153.2552
	16384	2444.166	2668.403	2410.417	2454.755	2433.651	2482.2784
	65536	38857.426	40814.289	39408.855	41416.488	40837.004	40266.8124
	262144	641947.438	677831.625	645831.241	638747.421	648231.217	650517.788
	1048576	?	?	?	?	?	
swap	1024	0.151	0.098	0.075	0.138	0.073	0.107
	4096	0.036	0.053	0.066	0.066	0.082	0.0606
	16384	1.073	1.538	1.181	1.801	1.31	1.3806
	65536	3.812	4.42	3.755	4.414	3.946	4.0694
	262144	14.541	15.753	15.008	14.718	15.107	15.0254
	1048576	56.5	58.559	61.605	57.582	57.657	58.3806

<표 8> Insertion sort 수행 시간

HEAP							
Data Format	Data Size	1	2	3	4	5	average
random	1024	0.451	0.552	0.513	0.519	0.378	0.4826
	4096	3.997	1.849	1.824	1.771	1.937	2.2756
	16384	9.219	9.034	9.305	7.429	8.576	8.7126
	65536	49.03	47.486	44.204	40.684	41.276	44.536
	262144	203.655	224.217	226.625	227.7	227.436	221.9266
	1048576	1027.759	1001.822	1000.164	1011.72	994.515	1007.196
descending	1024	0.321	0.611	0.32	0.313	0.33	0.379
	4096	1.672	1.601	2.055	1.805	1.741	1.7748
	16384	7.019	6.769	7.026	8.05	6.553	7.0834
	65536	36.126	34.301	33.212	31.925	33.277	33.7682
	262144	135.869	132.97	131.565	130.485	133.963	132.9704
	1048576	621.874	581.839	577.824	579.774	582.179	588.698
swap	1024	0.602	0.606	0.489	0.632	0.743	0.6144
	4096	0.351	0.687	0.485	0.436	0.45	0.4818
	16384	7.229	7.286	7.297	7.629	7.646	7.4174
	65536	32.831	31.878	31.915	32.117	32.999	32.348
	262144	166.598	135.508	137.698	136.822	136.969	142.719
	1048576	656.287	592.791	587.128	590.719	597.122	604.8094

<표 9> Heap sort 수행 시간

QSORT							
Data Format	Data Size	1	2	3	4	5	average
random	1024	1.03	0.897	0.69	0.911	0.82	0.8696
	4096	3.276	2.808	3.516	2.638	2.809	3.0094
	16384	12.314	12.335	12.362	12.091	12.512	12.3228
	65536	55.262	54.994	55.295	57.179	56.855	55.917
	262144	258.396	247.621	244.675	245.168	245.161	248.2042
	1048576	1136.205	1099.857	1092.714	1103.214	1110.089	1108.4158
descending	1024	0.407	0.481	0.337	0.582	0.373	0.436
	4096	1.402	1.272	1.48	1.411	1.308	1.3746
	16384	5.688	6.162	6.092	5.838	5.783	5.9126
	65536	24.55	26.357	25.362	26.378	25.504	25.6302
	262144	109.273	110.921	109.614	117.038	108.898	111.1488
	1048576	472.467	471.088	470.783	480.357	478.798	474.6986
swap	1024	0.335	0.417	0.375	0.618	0.301	0.4092
	4096	0.192	0.082	0.066	0.041	0.056	0.0874
	16384	5.095	6.191	5.662	5.04	4.785	5.3546
	65536	21.911	24.049	22.921	21.825	23.544	22.85
	262144	94.629	96.42	95.751	99.188	98.957	96.989
	1048576	425.948	438.292	428.98	445.479	442.298	436.1994

<표 10> Qsort 수행 시간

QUICK SORT							
Data Format	Data Size	1	2	3	4	5	average
random	1024	0.364	0.511	0.432	0.385	0.322	0.4028
	4096	1.397	1.376	1.521	1.345	1.57	1.4418
	16384	6.348	6.466	6.583	6.237	6.89	6.5048
	65536	27.444	27.995	27.305	26.494	29.907	27.829
	262144	124.541	125.072	124.403	120.714	126.351	124.2162
	1048576	561.628	552.842	554.041	578.11	565.842	562.4926
descending	1024	0.516	0.534	0.603	0.544	0.445	0.5284
	4096	1.969	2.199	2.327	1.945	2.113	2.1106
	16384	8.83	10.004	9.461	9.572	10.525	9.6784
	65536	43.814	49.426	42.665	44.798	45.949	45.3304
	262144	200.992	201.654	201.287	197.575	195.861	199.4738
	1048576	880.921	896.531	883.69	903.126	890.055	890.8646
swap	1024	0.241	0.288	0.274	0.339	0.3	0.2884
	4096	1.835	1.574	1.771	1.748	1.463	1.6782
	16384	5.011	4.805	5.098	4.687	4.651	4.8504
	65536	21.316	19.977	20.67	19.863	19.649	20.295
	262144	91.492	88.557	89.321	89.699	88.807	89.5752
	1048576	412.041	406.995	414.889	409.08	404.327	409.4664

<표 11> Quick sort 수행 시간

OPTIMIZED QUICK SORT							
Data Format	Data Size	1	2	3	4	5	average
random	1024	0.334	0.298	0.476	0.48	0.504	0.4184
	4096	1.044	1.056	1.031	1.017	1.056	1.0408
	16384	5.205	4.823	5.126	5.172	5.375	5.1402
	65536	25.027	20.82	21.084	22.175	21.821	22.1854
	262144	95.336	89.08	94.439	93.847	92.383	93.017
	1048576	394.634	384.611	377.703	379.343	381.879	383.634
descending	1024	0.348	0.292	0.385	0.298	0.304	0.3254
	4096	1.897	1.563	1.368	1.504	1.257	1.5178
	16384	6.227	5.873	5.786	5.916	5.687	5.8978
	65536	26.522	25.483	25.497	25.514	24.976	25.5984
	262144	114.102	111.311	110.524	112.541	111.644	112.0244
	1048576	513.646	493.745	508.925	525.303	493.452	507.0142
swap	1024	0.205	0.226	0.153	0.294	0.145	0.2046
	4096	1.655	1.491	1.574	1.577	1.777	1.6148
	16384	2.333	2.38	2.178	2.423	2.542	2.3712
	65536	10.393	10.238	9.858	11.438	11.012	10.5878
	262144	50.809	47.387	47.307	50.058	46.004	48.313
	1048576	226.386	228.92	254.113	236.307	228.15	234.7752

<표 12> Quick Sort Optimized 수행 시간

측정결과 분석-5)

Quick sort 와 Insertion sort 비교

INSERTION						
Size	1	2	3	4	5	average
8	0.004	0.017	0.005	0.005	0.003	0.0068
16	0.008	0.016	0.002	0.02	0.002	0.0096
32	0.006	0.026	0.006	0.006	0.015	0.0118
50	0.041	0.026	0.045	0.028	0.033	0.0346
64	0.024	0.049	0.025	0.036	0.023	0.0314
100	0.067	0.053	0.058	0.079	0.088	0.069
150	0.114	0.15	0.139	0.129	0.246	0.1556
200	0.211	0.316	0.372	0.211	0.211	0.2642
300	0.454	0.45	0.508	0.61	0.636	0.5316
400	0.927	0.795	0.781	0.807	0.799	0.8218
500	1.316	1.333	1.57	1.225	1.517	1.3922
600	1.776	2.067	2.4	1.823	1.671	1.9474

<표 13> Insertion Sort 의 작은 데이터 수행시간

QUICK						
Size	1	2	3	4	5	average
8	0.075	0.009	0.028	0.005	0.005	0.0244
16	0.004	0.007	0.003	0.003	0.056	0.0146
32	0.011	0.023	0.006	0.072	0.006	0.0236
50	0.01	0.012	0.01	0.017	0.01	0.0118
64	0.012	0.026	0.014	0.013	0.012	0.0154
100	0.02	0.023	0.033	0.023	0.057	0.0312
150	0.055	0.041	0.049	0.035	0.053	0.0466
200	0.05	0.091	0.081	0.083	0.079	0.0768
300	0.086	0.099	0.206	0.162	0.099	0.1304
400	0.105	0.178	0.22	0.196	0.22	0.1838
500	0.178	0.193	0.164	0.251	0.237	0.2046
600	0.224	0.161	0.243	0.203	0.217	0.2096

<표 14> Quick Sort 의 작은 데이터 수행시간