알고리즘설계와분석_Mp2_Mater Of Sorting Report

심리학과

20190345 김동현

Mp2 과제를 통해, 4 가지 정렬 알고리즘을 사용하여 정렬되지 않은 정수에 대한 정렬을 수행하였고, 각 알고리즘의 수행시간을 측정하여 비교하였다. 정렬 알고리즘은 삽입 정렬(insertion sort), 퀵 정렬(quick sort), 병합 정렬(merge sort), 최적화 정렬(optimization sort)이며, 최적화 정렬(optimization sort)는 퀵 정렬과 삽입 정렬을 혼합 사용하여 구현했다.

1. 삽입정렬, 퀵정렬, 병합정렬, 최적화 정렬의 구현 및 시간 복잡도 비교

4가지의 정렬 알고리즘의 구현 방법에 대해 소개한 후 랜덤으로 생성된 정수 sequence의 수행시간과 내림차순으로 생성된 정수 sequence의 수행시간을 비교하려 한다.

1) 정렬 알고리즘의 구현

우선 삽입 정렬은 아래 그림과 같은 코드로 구현하였으며, 시간 복잡도는 for(int i=start+1; i<=end; i++), while(j>=0&&arr[j]>key) 문의 실행 여부에 따라 달라진다. Best time complexity는 이미 정렬된 배열이 입력으로 들어오는 경우로, O(n)의 시간 복잡도를 갖는다. Worst time complexity는 내림차순으로 정렬된 배열이 입력으로 들어오는 경우로 $O(n^2)$의 시간 복잡도를 가진다.

```
void insertionSort(int* arr,int start,int end){
    for(int i=start+1;i<=end;i++){
        int key=arr[i];
        int j=i-1;
        while(j>=0&&arr[j]>key){
            arr[j+1]=arr[j];
            j--;
        }
        arr[j+1]=key;
    }
}
```

병합 정렬은 아래 그림과 같은 코드로 표현하였으며, 시간 복잡도는 mergeSort(int*,int,int) 함수의 재귀 호출 횟수에 따라 결정된다. mergeSort 함수는 배열을 2등분하여 배열의 원소의 개수가 1개가 될 때까지 재귀 호출된다. Merge(int*,int,int,int) 함수는 각각 정렬되어 있는 상태에 있는 두 배열을 합쳐 정렬된 하나의 배열을 만드는 함수로 O(n)의 시간 복잡도를 가진다. 이를 바탕으로 병합 정렬의 수행시간은 T(n)=2T(n/2)+O(n)이며, 시간복잡도는 $O(n \log n)$임을 알 수 있다.

```
void mergeSort(int* arr,int left, int right){
    if(left>=right) return;
    int pivot=(left+right)/2;
    mergeSort(arr,left,pivot);
    mergeSort(arr,pivot+1,right);
    merge(arr,left,pivot,right);
}

void merge(int* arr,int left, int pivot, int right){
    int nl=pivot-left+1;
    int nr=right-pivot;
    int i=0,j=0,k=left;
    int* larr=(int*)malloc(nl*sizeof(int));
    int* rarr=(int*)malloc(nr*sizeof(int));
    for(int idx=0;idx<nl;idx++){
        larr[idx]=arr[left+idx];
    }
    for(int idx=0;idx<nr;idx++){
        rarr[idx]=arr[pivot+idx+1];
    }
    while(i<nl&&j<nr){
        if(larr[i]<=rarr[j]) arr[k++]=larr[i++];
        else arr[k++]=rarr[j++];
    }
    while(i<nl){
        arr[k++]=larr[i++];
    }
    while(j<nr){
        arr[k++]=rarr[j++];
    }
    free(larr);
    free(rarr);
}
```

퀵 정렬은 아래 그림과 같은 코드로 구현하였으며, 퀵 정렬의 시간 복잡도는 quicksort(int*,int,int)의 재귀 호출 횟수에 따라 결정된다. Quicksort함수는 quickSortPartition함수의 반환값으로 정해지는 Pivot의 값에 따라 배열을 나누어 재귀 호출한다. 피벗값을 기준으로 배열을 나누어 정렬하는 quickSortPartition(int*,int,int,int)함수는 O(n)의 시간복잡도를 가진다. 이를 바탕으로 퀵 정렬의 시간 복잡도는 다음과 같다. 우선 average time complexity는 quickSortPartition 함수에 의

해 배열이 너무 치우치지 않게 나뉘는 경우로 T(n)=2T(n/2)+O(n)으로 O($n \log n$)임을 알 수 있다. 반면, worst time complexity는 quickSortPartition 함수에 의해 배열이 한 쪽으로 치우쳐 나뉘는 경우로 T(n)=T(n-1)+O(n)의 수행시간을 보이며 O(n^2)의 시간 복잡도를 가진다.

```
void quickSort(int* arr,int left, int right){
    if(left<right){
        int pivot=quickSortPartition(arr,left,right,right);
        quickSort(arr,left,pivot-1);
        quickSort(arr,pivot+1,right);
    }
}

int quickSortPartition(int* arr,int left, int right,int pivot){
    int i=left-1;
    int tmp=arr[pivot];
    arr[pivot]=arr[right];
    arr[right]=tmp;
    int pivot_dat=arr[right];
    for(int j=left;j<right;j++){
        if(arr[j]<=pivot_dat){
            i++;
            tmp=arr[i];
            arr[i]=arr[j];
            arr[j]=tmp;
        }
    }
    tmp=arr[i+1];
    arr[i+1]=arr[right];
    arr[right]=tmp;
    return i+1;
}
```

최적화 함수는 기존 함수인 삽입 정렬과 퀵 정렬을 혼합하여 새로운 정렬 함수를 구현하였다. 정렬하려는 원소의 개수가 일정 개수 이하가 되면, 해당 배열에 대해 삽입 정렬을 실시한다. 뿐만 아니라 opti_partition(int*,int,int) 함수를 통해 배열 내의 5개의 원소 중 중앙값을 피벗으로 설정. 이로 인해 quickSort의 average time complexity인 O($n \log n$)을 유지하면서, worst time complexity가 O(n^2)이 되는 경우를 예방하고자 한다.

```c
void optimizationSort(int* arr,int left,int right){
    if(right-left<=RUN){
        insertionSort(arr,left,right);
    }
    else{
        int pivot=opti_partition(arr,left,right);
        optimizationSort(arr,left,pivot-1);
        optimizationSort(arr,pivot+1,right);
    }
}

int opti_partition(int* arr,int left, int right){
    int pivot_list[5]={left,left+(right-left)/4,left+(right-left)/2,left+(3*(right-left))/4,right};
    for(int i=0;i<3;i++){
        int max=0;
        for(int j=1;j<5-i;j++){
            if(arr[pivot_list[j]]>arr[pivot_list[max]]){
                max=j;
            }
        }
        int tmp=pivot_list[max];
        pivot_list[max]=pivot_list[4-i];
        pivot_list[4-i]=tmp;
    }
    return quickSortPartition(arr,left,right,pivot_list[2]);
}
```

2) 정렬 알고리즘의 시간 측정 비교

위에서 설명한 4가지 코드를 실행하여 다양한 입력 개수에 따른 정렬을 실시하였다. 평균 시간을 측정하기 위해 각각의 입력 개수마다 10개의 입력 파일을 만들었으며, 입력파일은 c언어의 rand()함수를 이용해 정수를 생성하였다.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM 1073741824
#define RANGE 1073741824
#define FILELEN 30

int main(){

    char filename[FILELEN];
    printf("file name : ");
    scanf("%s",filename);
    srand(time(NULL));
    FILE* fp=fopen(filename,"wt");

    fprintf(fp,"%d ",NUM);
    for(int i=0;i<NUM;i++){
        int num=rand()%RANGE-(RANGE/2);
        fprintf(fp,"%d ",num);
    }

    fclose(fp);


    return 0;
}
```

입력 개수가 4(2^2)개인 경우에 대해서 정렬 방법에 따른 수행시간을 정리하면 다음과 같다. 삽입 정렬은 0.000024초, 퀵 정렬은 0.0000031초, 병합 정렬은 0.0000047초, 최적화 정렬은 0.0000031초로 정렬 방법 간 유의미한 시간 차이를 보이지 않는다.

n4	case1	case2	case3	case4	case5	case6	case7	case8	case9	case10	average
1(insertion)	0.000003	0.000002	0.000002	0.000003	0.000002	0.000002	0.000003	0.000002	0.000002	0.000003	0.0000024
2(quick)	0.000003	0.000003	0.000004	0.000004	0.000003	0.000003	0.000003	0.000003	0.000002	0.000003	0.0000031
3(merge)	0.000004	0.000004	0.000005	0.000005	0.000005	0.000005	0.000005	0.000004	0.000005	0.000005	0.0000047
4(opti)	0.000003	0.000003	0.000004	0.000003	0.000003	0.000003	0.000004	0.000003	0.000002	0.000003	0.0000031

입력 개수가 64(2^6)개인 경우에 대해서 정렬 방법에 따른 수행시간을 정리하면 다음과 같다. 삽입 정렬은 0.0000145초, 퀵 정렬은 0.0000128초, 병합 정렬은 0.0000305초, 최적화 정렬은 0.0000112초로 정렬 방법 간 유의미한 시간 차이를 보이지 않는다.

n64	case1	case2	case3	case4	case5	case6	case7	case8	case9	case10	average
1(insertion)	0.000013	0.000013	0.000013	0.000026	0.000014	0.000013	0.000014	0.000013	0.000013	0.000013	0.0000145
2(quick)	0.000013	0.000008	0.000012	0.000012	0.000012	0.000014	0.000014	0.000014	0.000015	0.000014	0.0000128
3(merge)	0.000028	0.000031	0.000029	0.000028	0.000032	0.00004	0.000028	0.00003	0.000029	0.00003	0.0000305
4(opti)	0.000011	0.000012	0.000011	0.000012	0.000012	0.000011	0.000011	0.000011	0.00001	0.000011	0.0000112

입력 개수가 1024(2^{10})개인 경우에 대해서 정렬 방법에 따른 수행시간을 정리하면 다음과 같다. 삽입 정렬은 0.0020286초, 퀵 정렬은 0.0002426초, 병합 정렬은 0.0004839초, 최적화 정렬은 0.0002224초이다. 입력 개수가 증가할수록 삽입 정렬의 수행시간이 나머지 정렬 방법보다 10배 차이가 나는 것을 확인할 수 있다.

n1024	case1	case2	case3	case4	case5	case6	case7	case8	case9	case10	average
1(insertion)	0.002026	0.002003	0.002052	0.002059	0.002059	0.002083	0.001929	0.001834	0.001958	0.002283	0.0020286
2(quick)	0.00025	0.000258	0.000223	0.000258	0.000246	0.000244	0.000242	0.00023	0.000253	0.000222	0.0002426
3(merge)	0.00051	0.000477	0.000493	0.000453	0.000513	0.000455	0.000451	0.000494	0.000491	0.000502	0.0004839
4(opti)	0.00022	0.000224	0.000221	0.000221	0.000219	0.000216	0.000245	0.000221	0.000219	0.000218	0.0002224

입력 개수가 16384(2^{14})개인 경우에 대해서 정렬 방법에 따른 수행시간을 정리하면 다음과 같다. 삽입 정렬은 0.3021073초, 퀵 정렬은 0.0052428초, 병합 정렬은 0.0094767초, 최적화 정렬은 0.0046942초이다. 삽입 정렬의 수행시간이 나머지 정렬 방법보다 30~60배 차이가 나는 것을 확인할 수 있다. 앞선 경우와 비교하자면 삽입정렬의 증가 정도가 다른 정렬 방법에 비해 더 가파르게 상승하는 것을 알 수 있다. 뿐만 아니라 병합 정렬의 경우에도 퀵 정렬과 퀵정렬을 기반으로 한 최적화 정렬에 비해 2배 가까이 수행시간이 증가한 것을 알 수 있다.

n16384	case1	case2	case3	case4	case5	case6	case7	case8	case9	case10	average
1(insertion)	0.299784	0.301207	0.302894	0.301005	0.304756	0.301957	0.303021	0.300689	0.303378	0.302382	0.3021073
2(quick)	0.005222	0.0054	0.004652	0.005272	0.005375	0.00527	0.005307	0.005297	0.005367	0.005266	0.0052428
3(merge)	0.008341	0.009743	0.00894	0.00974	0.009654	0.00977	0.009644	0.009783	0.009379	0.009773	0.0094767
4(opti)	0.004388	0.004389	0.004748	0.004772	0.004779	0.004746	0.004809	0.004764	0.004765	0.004782	0.0046942

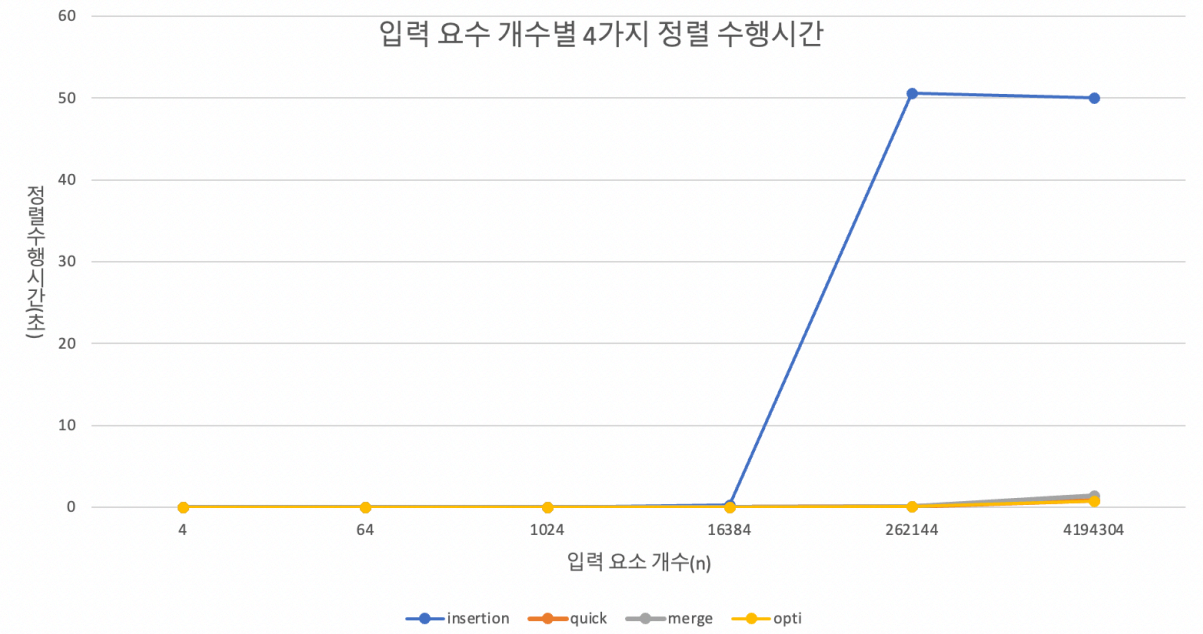
입력 개수가 262144(2^{18})개인 경우에 대해서 정렬 방법에 따른 수행시간을 정리하면 다음과 같다. 삽입 정렬은 50.6295374초, 퀵 정렬은 0.0646991초, 병합 정렬은 0.1104118초, 최적화 정렬은 0.0608003초이다. 삽입 정렬의 수행시간이 나머지 정렬 방법보다 500~1000배 차이가 나는 것을 확인할 수 있다. 앞선 경우와 비교하자면 삽입정렬의 증가 정도가 다른 정렬 방법에 비해 더 가파르게 상승하는 것을 알 수 있다. 뿐만 아니라 병합 정렬의 경우에도 퀵 정렬과 퀵정렬을 기반으로 한 최적화 정렬에 비해 2배 가까이 수행시간이 증가한 것을 알 수 있다.

n262144	case1	case2	case3	case4	case5	case6	case7	case8	case9	case10	average
1(insertion)	50.609689	50.673078	50.583308	50.555459	50.653116	50.578092	50.696753	50.822743	50.610313	50.512823	50.6295374
2(quick)	0.066563	0.063876	0.064863	0.064653	0.064391	0.064309	0.064205	0.063672	0.063543	0.066916	0.0646991
3(merge)	0.11222	0.110625	0.110567	0.111797	0.109824	0.11072	0.108655	0.11071	0.108449	0.110551	0.1104118
4(opti)	0.060401	0.05947	0.062204	0.060376	0.060056	0.059841	0.060555	0.06195	0.060588	0.062562	0.0608003

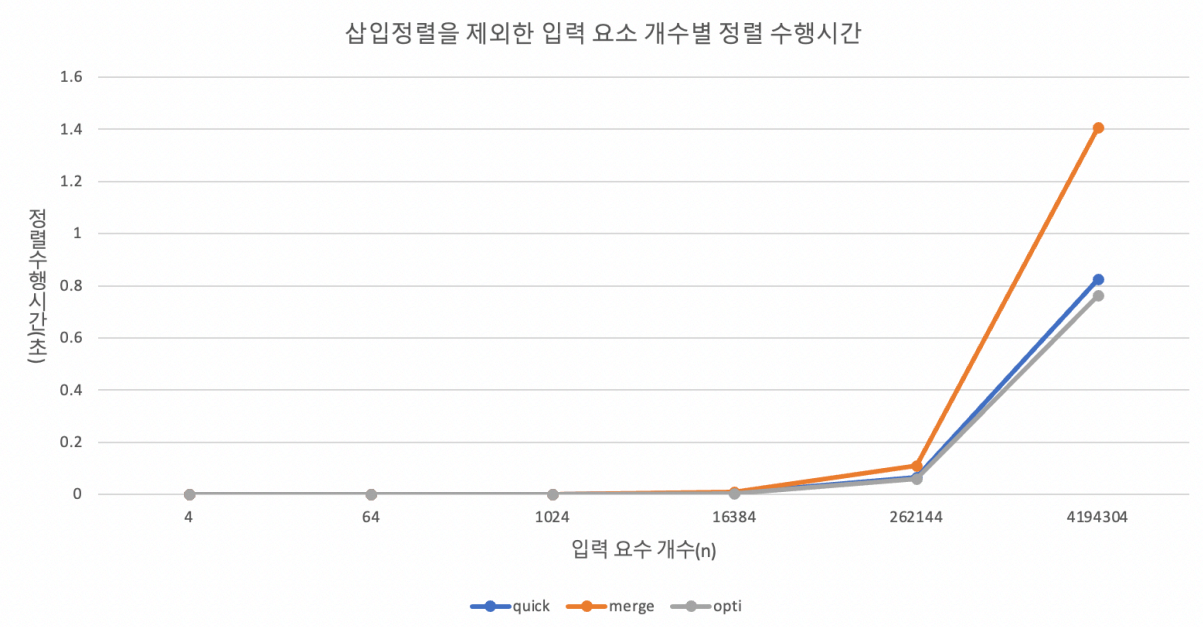
입력 개수가 4194304(2^{22})개인 경우에 대해서 정렬 방법에 따른 수행시간을 정리하면 다음과 같다. 삽입 정렬은 cspro환경 상에서 수행되지 못할 정도로 시간이 많이 걸리는 모습을 확인했다. 반면, 퀵 정렬은 0.8268305초, 병합 정렬은 1.4088759초, 최적화 정렬은 0.7634802초이다. 병합 정렬의 경우에도 퀵 정렬과 퀵정렬을 기반으로 한 최적화 정렬에 비해 2배 가까이 수행시간이 증가한 것을 알 수 있다. 뿐만 아니라 최적화 정렬 방법이 다른 정렬 방법에 비해 빠른 수행시간을 보이고 있다. 이는 이전 경우보다 차이가 더 나타나는 것으로 보아 최적화 정렬이 입력 파일 구성에 따른 차이와는 독립적으로 수행능력이 좋다는 것을 확인할 수 있다.

n4194304	case1	case2	case3	case4	case5	case6	case7	case8	case9	case10	average
1(insertion)											
2(quick)	0.841013	0.821723	0.824414	0.81841	0.816145	0.863031	0.810421	0.819261	0.837782	0.816105	0.8268305
3(merge)	1.417111	1.404448	1.407311	1.40614	1.408959	1.404931	1.413671	1.404545	1.410165	1.411478	1.4088759
4(opti)	0.833761	0.745247	0.756293	0.74646	0.777925	0.754437	0.756046	0.755013	0.751115	0.758505	0.7634802

이러한 입력 개수에 따른 정렬 방법 간 수행 시간을 그래프화하면 다음과 같다.



삽입 정렬이 타 정렬에 비해 빠른 속도로 증가하는 것을 알 수 있다. 삽입정렬을 제외한 나머지 정렬 방법간 수행 시간을 비교하기 위해 삽입 정렬을 제외한 정렬 방법 간 수행 시간을 그래프화 하였다.



병합 정렬이 입력의 크기가 증가할수록 수행시간이 더 걸리는 모습을 확인할 수 있다. 또한 최적화 정렬이 퀵 정렬에 비해 입력의 크기가 증가할수록 수행시간이 줄어드는 모습을 확인할 수 있다.

요약하자면, 4가지 정렬방법의 평균 수행 시간은 삽입정렬, 병합정렬, 퀵정렬, 최적화정렬 순으로 빨라지는 것을 알 수 있다. 특히 삽입정렬은 시간 복잡도가 $O(n^2)$으로 $O(n \log n)$의 시간복잡도를 가지는 나머지 정렬 방식에 비해 입력 개수가 증가할수록 빠른 속도로 수행 시간이 증가함을 알 수 있다. 병합정렬과 퀵정렬, 최적화 정렬은 $O(n \log n)$의 시간복잡도를 가지기 때문에 asympototic한 입력 개수 증가에 따른 수행 시간 차이를 보이지는 않으나, constant의 차이로 인해 약간씩의 차이를 보이고 있다.

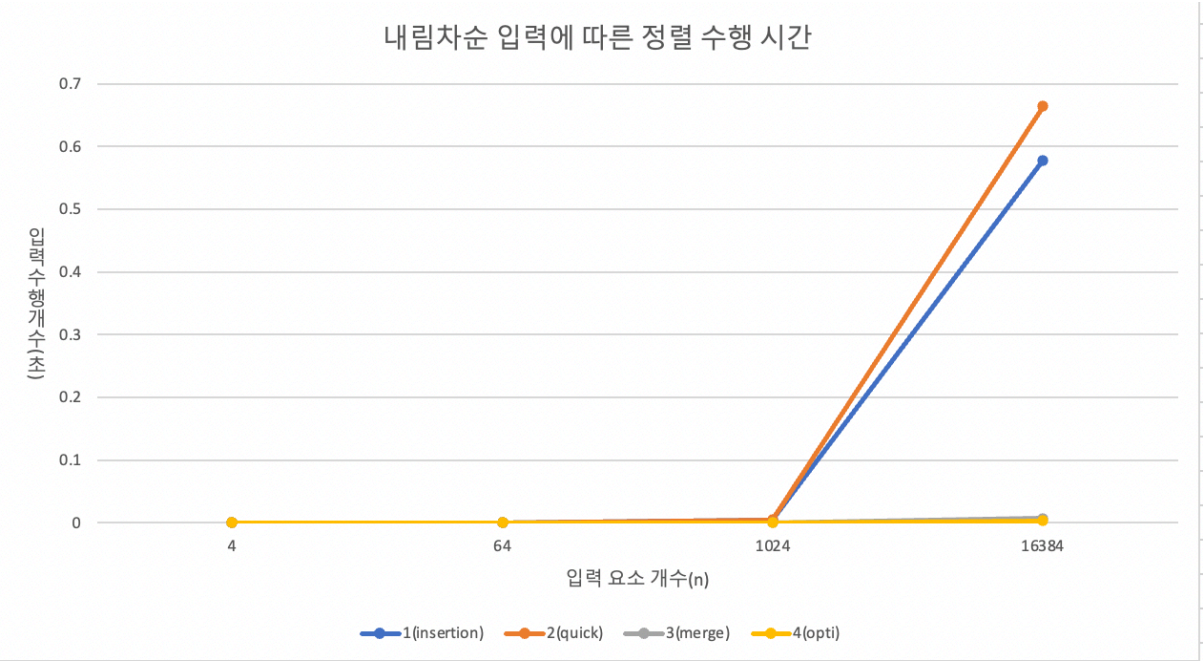
위에서 설명한 4가지 코드를 바탕으로 내림차순으로 정렬된 입력에 대해서 정렬을 실시하였다. 입력파일은 c언어를 이용하여 입력 크기에 따라 내림차순으로 정수를 생성하여 파일 출력을 실행하였다.

```c
#include <stdio.h>
#include <stdlib.h>

#define FILELEN 30

int main(){
    int num;
    char filename[FILELEN];
    printf("file name : ");
    scanf("%s",filename);
    printf("num : ");
    scanf("%d",&num);
    FILE* fp=fopen(filename,"wt");

    fprintf(fp,"%d ",num);
    for(int i=num;i>0;i--){
        fprintf(fp,"%d ",i);
    }

    fclose(fp);


    return 0;
}
```

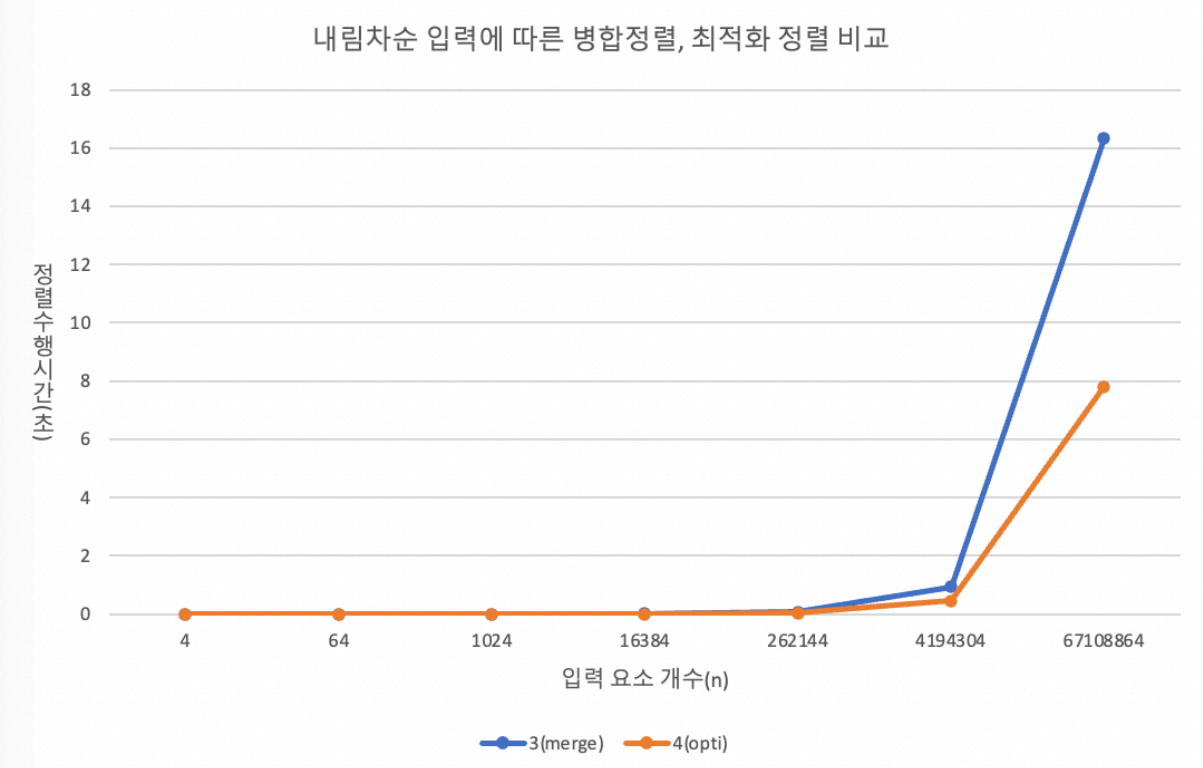
입력 크기에 따른 4가지 정렬 방법의 수행시간은 아래 표와 같다.

	4	64	1024	16384	262144	4194304	67108864	1073741824
1(insertion)	0.000003	0.00002	0.004235	0.578063	101.151136	0		0
2(quick)	0.000003	0.000026	0.004693	0.664196	0	0	0	0
3(merge)	0.000004	0.000025	0.000322	0.006655	0.078527	0.927192	16.326397	0
4(opti)	0.000003	0.000014	0.00017	0.003027	0.040287	0.450346	7.806884	0

삽입 정렬과 퀵 정렬은 내림차순 입력에 대해 $O(n^2)$의 시간 복잡도를 보이고 있다. 삽입 정렬은 모든 입력에 대해 두번의 반복문을 실행해야하기 때문에 랜덤 입력에 따른 정렬 방법보다 더 많은 시간이 소요되는 것을 보인다. 퀵 정렬 또한 partition이 한쪽으로 치우치는 불균형한 재귀 호출을 호출하기 때문에 랜덤 입력에 따른 정렬 방법보다 더 많은 시간이 소요되는 것을 보인다. 반면, 병합정렬은 기존 랜덤 입력에 따른 정렬 방법과 비슷한 수행 시간을 보이고 $O(n \log n)$의 시간 복잡도를 가지고 있다. 최적화 정렬 역시 기존 랜덤 입력에 따른 정렬 방법과 비슷한 수행 시간을 보이고 $O(n \log n)$의 시간 복잡도를 가져, 퀵정렬을 기반으로 정렬을 수행하지만, 퀵 정렬이 worst case 시간 복잡도의 한계를 보완한 모습을 보인다. 뿐만 아니라 입력 크기가 일정 크기 이상을 넘어가면 퀵 정렬, 삽입정렬, 병합정렬, 최적화정렬 순으로 cspro상에서 수행되지 않는 모습을 보인다. 이를 그래프로 표현하면 다음과 같다.



삽입정렬과 퀵 정렬이 비슷한 모양으로 증가하는 것과 병합 정렬과 최적화 정렬이 비슷한 모양으로 증가하는 것을 알 수 있다.



병합 정렬과 최적화 정렬을 비교하자면, 입력 개수가 적을 때에는 유의미한 차이를 보이지 않지만, 입력 개수가 증가함에 따라 최적화 정렬이 보다 빠른 수행 시간을 가지는 양상을 보인다.

요약하자면, 내림차순 입력에 대한 4가지 정렬방법의 수행 시간은 퀵정렬, 삽입정렬, 병합정렬, 최적화정렬 순으로 빨라지는 것을 알 수 있다. 특히 삽입정렬과 퀵정렬은 시간 복잡도가

$O(n^2)$으로 $O(n \log n)$의 시간복잡도를 가지는 병합정렬과 최적화정렬 방식에 비해 입력 개수가 증가할수록 빠른 속도로 수행 시간이 증가함을 알 수 있다. 퀵정렬은 평균 수행시간에서는 $O(n \log n)$의 시간 복잡도를 가지지만 내림차순 정렬은 worst case로서 $O(n^2)$의 시간복잡도를 가진다. 병합정렬과 최적화 정렬은 $O(n \log n)$의 시간복잡도를 가지기 때문에 asympototic한 입력 개수 증가에 따른 수행 시간 차이를 보이지는 않으나, constant의 차이로 인해 약간씩의 차이를 보이고 있다.

2. 최적화 정렬의 구현

앞선 3가지 알고리즘은 다음과 같은 문제점이 있다. 삽입 정렬은 $O(n^2)$의 시간복잡도를 가지기 때문에 입력의 크기가 증가할수록 수행시간이 오래 걸린다. 병합 정렬은 $O(n \log n)$의 시간 복잡도를 가지며, 비교 정렬 중 가장 최적화된 시간 복잡도를 가지고 있다. 하지만 퀵정렬에 비해 constant값이 크기 때문에 같은 $O(n \log n)$시간 복잡도임에도 입력의 크기가 늘어남에 따라 수행시간이 늘어난다. 또한 재귀 호출을 할때마다 새로운 배열을 할당하고 복사하기 때문에 메모리 사용량 또한 늘어난다. 퀵 정렬은 $O(n \log n)$의 시간 복잡도를 가지지만 worst case의 경우 $O(n^2)$의 시간 복잡도를 가진다. 실제 내림차순 입력에 대해서는 삽입정렬의 시간복잡도와 유사한 모습을 보인다. 이러한 각 정렬의 문제점을 보완하여 향상된 정렬 알고리즘을 구현하기 위해, 아래 코드를 구현하였다.

```
void optimizationSort(int* arr,int left,int right){
    if(right-left<=RUN){
        insertionSort(arr,left,right);
    }
    else{
        int pivot=opti_partition(arr,left,right);
        optimizationSort(arr,left,pivot-1);
        optimizationSort(arr,pivot+1,right);
    }
}

int opti_partition(int* arr,int left, int right){
    int pivot_list[5]={left,left+(right-left)/4,left+(right-left)/2,left+(3*(right-left))/4,right};
    for(int i=0;i<3;i++){
        int max=0;
        for(int j=1;j<5-i;j++){
            if(arr[pivot_list[j]]>arr[pivot_list[max]]){
                max=j;
            }
        }
        int tmp=pivot_list[max];
        pivot_list[max]=pivot_list[4-i];
        pivot_list[4-i]=tmp;
    }
    return quickSortPartition(arr,left,right,pivot_list[2]);
}
```

우선 기존에 작성한 퀵 정렬을 활용하였다. 퀵 정렬은 평균 시간 복잡도가 $O(n \log n)$이며, 이는 비교 정렬 알고리즘 중 최선의 시간 복잡도이다. 뿐만 아니라 앞선 병합 정렬과의 비교에서도 퀵 정렬이 근소하게 빠른 모습을 보였다. 퀵 정렬에서 적은 수의 원소를 정렬하기 위해 함수

를 재귀호출 하는 과정에서 발생하는 비효율성과 worst case에서 가지는 O(n^2)의 시간복잡도를 보완하기 다음과 같은 최적화 알고리즘을 구현하였다. 첫번째 문제인 적은 입력 수에 대한 불필요한 재귀 호출 대신 RUN 변수를 선언하여 RUN보다 적은 원소를 정렬할 경우, 퀵 정렬 호출이 아닌 삽입 정렬을 호출하여 재귀 호출 횟수를 줄인다. 삽입 정렬의 시간 복잡도가 O(n^2)이지만, 입력개수가 적은 경우 더 느리다고 볼 수 없으며, 오히려 재귀 호출을 줄여 시간 복잡도를 감소할 수 있을 것이다. 두번째 문제인 최악의 경우에 대한, 혹은 최악에 가까운 경우에 대한 입력을 보다 균형있게 정렬하기 위해, 새로운 partition함수를 정의하였다. 이는 가장 오른쪽에 있는 수를 pivot으로 설정하여 partition을 수행하는 기존 퀵 정렬의 방식이 아닌 새로운 pivot을 설정하여 partition을 진행하였다. Pivot을 정하는 규칙은 다음과 같다. 가장 왼쪽 원소, 1/4지점에 있는 원소, 1/2지점에 있는 원소, 3/4지점에 있는 원소, 가장 오른쪽 원소, 총 5가지의 원소 중 중앙값인 3번째로 큰 원소를 정한다. 이러한 방식으로 pivot을 정하는 방식은 pivot값을 정하는 과정에서 시간이 소비되고, 5개 중 중앙값 역시 한쪽에 급격히 치우친 값일 수 있다는 위험성이 존재한다. 하지만 가장 오른쪽 원소만을 고르는 pivot이 치우쳐진 pivot이 될 확률에 비해 낮은 확률을 가지고 있고, 이러한 확률의 감소를 위해 소비하는 시간이 보다 더 효율적이기에 해당 방법으로 pivot을 정하였다.

3. 실험 환경

해당 실험은 다음과 같은 환경에서 실행하였다.

Model : MacBook Air
Cpu : APPLE M2
Os type : macOS Monterey
Os version : 12.6 version
Memory size : 16GB

4. 실험 설정

해당 실험에서는 자료를 저장하기 위해 배열을 사용하였다. 입력 파일에서 원소의 개수를 받아들여 해당 개수에 맞는 배열을 동적 할당을 통해 생성하였으며, 자료의 사용이 완료된 후 동적 해제를 통해 메모리를 반환하였다. 뿐만 아니라 병합 정렬에서 사용하는 배열 역시 동적 할당을 통해 생성한 배열로, 배열의 사용이 완료된 후 동적 해제를 통해 메모리를 반환하였다.

평균 시간 복잡도를 측정하기 위해 정수형 배열을 사용하였는데, 이 정수의 범위는 –(입력개수*10/2)부터 +(입력개수*10/2)-1이다. 예를 들어 원소의 개수가 64개인 입력 데이터의 범위는 -320 ~ +319이다. 중복을 줄이기 위해 입력 개수에 10을 곱하였으며, 음수를 포함하기 위해 범위/2를 rand함수에서 생성한 값에서 뺐다. 반면, 내림차순으로 정렬된 배열 역시 정수형 배열을 사용하였으며, 이 정수의 범위는 1부터 원소의 개수까지 설정하였다. 예를 들어 원소의 개수가 64인 입력 데이터의 범위는 1 ~ 64로, 64,63,…,2,1의 순서로 배열에 저장된다.

5. 코멘트

삽입정렬, 병합정렬, 퀵정렬, 최적화 정렬 4가지 알고리즘을 구현하였다. 삽입정렬의 시간복잡도는 O(n^2), 병합정렬의 시간복잡도는 O($n \log n$),퀵정렬의 시간 복잡도는 average input case의 경우에는 O($n \log n$), worst input case의 경우에는 O(n^2), 최적화 정렬의 시간 복잡도는 O($n \log n$)을 보인다. 이를 비교하기 위해 c언어의 rand함수를 사용하여 랜덤으로 정수를 생성하여 입력 개수

에 따른 정렬 수행 시간을 알고리즘 별로 비교하였다. O(n^2)의 시간복잡도를 가진 삽입 정렬은 입력 개수가 늘어남에 따라 수행시간이 극적으로 증가하였으며, O($n \log n$)의 시간복잡도를 가진 병합 정렬, 퀵 정렬, 최적화 정렬은 삽입 정렬에 비해 상대적으로 수행시간이 크게 증가하지 않았다. 그러나 3가지 정렬 내에서도 수행시간 간의 차이는 존재하였고, 병합정렬, 퀵정렬, 최적화 정렬 순으로 수행시간이 감소했다. 또한 내림차순으로 생성된 정수를 입력 개수에 따라 정렬 수행시간을 알고리즘별로 비교하였다. O(n^2)의 시간복잡도를 가진 삽입 정렬과 퀵정렬은 입력 개수가 늘어남에 따라 수행시간이 극적으로 증가하였으며, O($n \log n$)의 시간복잡도를 가진 병합 정렬과 최적화 정렬은 삽입 정렬에 비해 상대적으로 수행시간이 크게 증가하지 않았다.

최적화 정렬을 기존의 정렬 알고리즘을 활용하여 구현했다. 퀵 정렬을 기반으로 최적화 정렬을 구현하였으며, 퀵 정렬의 단점을 보완하고자 했다. 원소의 개수가 적을 때에도 재귀호출을 하는 퀵정렬의 비효율성을 보완하기 위해, 일정 개수 이하의 정렬에 대해서는 재귀 호출을 하지 않는 삽입 정렬을 호출하였다. 뿐만 아니라 partition과정에서 결정되는 pivot에 따라 불균형하게 배열이 나뉘어 재귀호출의 개수가 증가하는 비효율성을 보완하기 위해, 여러개의 pivot 후보 중 중앙값을 pivot으로 설정하도록 하여, 불균형하게 배열이 나뉘는 것을 예방하고자 했다.

정렬 알고리즘을 구현하는 것에 그치지 않고, 다양한 입력 케이스를 통해 수행시간을 정리하고 비교하였다. 이를 통해 입력 개수에 따른 정렬 수행 시간의 변화 뿐 아니라 같은 입력 개수에 따른 정렬 간 수행 시간을 비교하며, Big-O notation으로만 알고 있던 시간 복잡도를 직접 확인할 수 있었다. 이러한 비교를 통해 보다 시간적으로 효율적인 알고리즘을 구현하였고, 일반적인 입력 경우 뿐만 아니라 최악의 입력 경우에서도 시간적으로 효율적인 알고리즘을 구현하도록 했다.