whw4

B911012 금예인

1. (a) Selection Sort, Insertion Sort, Merge Sort, Quicksort의 n 값에 따른 C_1(the number of key comparisons)과 (b) C_2(the number of writing operations on the array to be sorted)는 프로그램 실행 결과 다음과 같았다.

```
[B911012@localhost whw4]$ sorting-comparisons<onemillion.in

** Selection Sort **
n = 100; C1 = 4950; C2 = 198.
n = 200; C1 = 19900; C2 = 398.
n = 300; C1 = 44850; C2 = 598.
n = 400; C1 = 79800; C2 = 798.
n = 500; C1 = 124750; C2 = 998.
n = 600; C1 = 179700; C2 = 1198.
n = 700; C1 = 244650; C2 = 1398.
n = 800; C1 = 319600; C2 = 1598.
n = 900; C1 = 404550; C2 = 1798.

** Insertion Sort **
n = 100; C1 = 2643; C2 = 2742.
n = 200; C1 = 10507; C2 = 10706.
n = 300; C1 = 21685; C2 = 21984.
n = 400; C1 = 39027; C2 = 39426.
n = 500; C1 = 63237; C2 = 63736.
n = 600; C1 = 88097; C2 = 88696.
n = 700; C1 = 113886; C2 = 114585.
n = 800; C1 = 160128; C2 = 160927.
n = 900; C1 = 198991; C2 = 199890.

** Merge Sort **
n = 100; C1 = 535; C2 = 1344.
n = 200; C1 = 1290; C2 = 3088.
n = 300; C1 = 2096; C2 = 4976.
n = 400; C1 = 2965; C2 = 6976.
n = 500; C1 = 3865; C2 = 8976.
n = 600; C1 = 4796; C2 = 11152.
n = 700; C1 = 5767; C2 = 13352.
n = 800; C1 = 6691; C2 = 15552.
n = 900; C1 = 7741; C2 = 17752.

** Quick Sort **
n = 100; C1 = 579; C2 = 352.
n = 200; C1 = 1621; C2 = 730.
n = 300; C1 = 3235; C2 = 1202.
n = 400; C1 = 4050; C2 = 1638.
n = 500; C1 = 5507; C2 = 2118.
n = 600; C1 = 6276; C2 = 2688.
n = 700; C1 = 6865; C2 = 3206.
n = 800; C1 = 31486; C2 = 3634.
n = 900; C1 = 32272; C2 = 4236.
```

- 프로그램 코드는 다음과 같다.

```
#include <stdio.h>

#define NN 25600  // big enough for buffer in mergesort and data0.
int C1, C2;  // number of comparisons and writings

void selectionsort(int arr[], int n)
{
    int i = 0;
    int k = 0;
    int min = arr[0];
    int* q = NULL;;
    int tmp = 0;;

    for (k = 0; k < n - 1; k++) {
        min = arr[k];
        for (i = k + 1; i < n; i++) {
            if (min > arr[i]) {
                min = arr[i];
                q = &arr[i];
            }
            C1++;
        }
        tmp = arr[0];
        arr[0] = min;
        *q = tmp;
        C2 += 2;
    }
}

void insertionsort(int arr[], int n) {
    int i = 0;
    int j = 1;
    int key = 0;

    for (j = 1; j < n; j++) {
        key = arr[j];
        for (i = j - 1; i >= 0; i--) {
            if (arr[i] > key) {
                C1++;
                arr[i + 1] = arr[i];
                C2++;
```

```
            }
            else {
                break;
            }
        }
        arr[i + 1] = key;
        C2++;
    }
}

void merge(int arr[], int l, int m, int r) // l<m<r
{
    int b[NN + 1];  // buffer to store two sublists.
    int i, j, k;

    // copy sublists to buffer
    for (i = l; i <= r; i++)
    {
        b[i] = arr[i];   C2++;
    }

    i = l; j = m + 1; k = l; // reset indices.

    // merge sublists in buffer into arr[]
    while (i <= m && j <= r) {
        if (b[i] <= b[j])
        {
            arr[k++] = b[i++]; C2++;
        }
        else
        {
            arr[k++] = b[j++]; C2++;
        }
        C1++;
    }
    if (i > m)
        while (j <= r)
        {
            arr[k++] = b[j++]; C2++;
        }
    else
        while (i <= m)
        {
```

```
            arr[k++] = b[i++]; C2++;
        }
}

void mergesort(int arr[], int l, int r)
{
    int m;

    if (l < r) {
        m = (l + r) / 2;
        mergesort(arr, l, m);
        mergesort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}


int partition(int arr[], int l, int r) {
    int x = arr[l];
    int i = l + 1;
    int j = r;
    int tmp = 0;

    while (1) {
        while (arr[i] <= x) {
            i++;
            C1++;
        }
        while (arr[j] > x) {
            j--;
            C1++;
        }
        if (i < j) {
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            C2 += 2;
        }
        else
            break;
    }
    tmp = arr[l];
    arr[l] = arr[j];
```

```
    arr[j] = tmp;
    C2 += 2;

    return j;
}

void quicksort(int arr[], int l, int r) {
    int p = 0;

    if (l < r) {
        p = partition(arr, l, r);
        quicksort(arr, l, p - 1);
        quicksort(arr, p + 1, r);
    }
}

int main()
{
    int i, j;
    int data0[NN];

    int N[10];
    N[1] = 100;
    N[2] = 200;
    N[3] = 300;
    N[4] = 400;
    N[5] = 500;
    N[6] = 600;
    N[7] = 700;
    N[8] = 800;
    N[9] = 900;

    printf("\n** Selection Sort **\n");
    for (i = 1; i < 10; i++) {
        // read N[i] data
        for (j = 0; j < N[i]; j++) {
            scanf("%d", &data0[j]); // read the input
        }
        C1 = C2 = 0;
        selectionsort(data0, N[i]);
        printf("n = %d; C1 = %d; C2 = %d.\n", N[i], C1, C2);
    }
```

```
    printf("\n** Insertion Sort **\n");
    for (i = 1; i < 10; i++) {
        // read N[i] data
        for (j = 0; j < N[i]; j++) {
            scanf("%d", &data0[j]); // read the input
        }
        C1 = C2 = 0;
        insertionsort(data0, N[i]);
        printf("n = %d; C1 = %d; C2 = %d.\n", N[i], C1, C2);
    }

    printf("\n** Merge Sort **\n");
    for (i = 1; i < 10; i++) {
        // read N[i] data
        for (j = 0; j < N[i]; j++) {
            scanf("%d", &data0[j]); // read the input
        }
        C1 = C2 = 0;
        mergesort(data0, 0, N[i] - 1);
        printf("n = %d; C1 = %d; C2 = %d.\n", N[i], C1, C2);
    }

    printf("\n** Quick Sort **\n");
    for (i = 1; i < 10; i++) {
        // read N[i] data
        for (j = 0; j < N[i]; j++) {
            scanf("%d", &data0[j]); // read the input
        }
        C1 = C2 = 0;
        quicksort(data0, 0, N[i] - 1);
        printf("n = %d; C1 = %d; C2 = %d.\n", N[i], C1, C2);
    }
}
```

2. C_1의 값은 네 정렬 방식 모두 n이 커질수록 증가하였으며, n이 같은 값일 때 정렬 방식 간의 C_1 대소 관계는 다음과 같다.

Merge Sort < Quick Sort < Insertion Sort < Selection Sort

C_2의 값 역시 네 정렬 방식 모두 n이 커질수록 증가하였으며, n이 같은 값일 때 정렬 방식 간의 C_2 대소 관계는 다음과 같다.

Selection Sort < Quick Sort < Merge Sort < Insertion Sort

C_1과 C_2 모두, 작을수록 performance가 뛰어나다고 볼 수 있다.

(a) Insertion Sort의 C_1(the number of key comparisons)은 Selection Sort보다 작았으나, C_2(the number of writing operations on the array to be sorted)는 Selection Sort보다 큰 값을 보였다. 따라서, 전반적인 performance가 항상 Selection Sort보다 뛰어나다고 말할 수는 없다.

(b) Quick Sort running time의 worst-case($O(n^2)$)는 데이터 값들이 이미 정렬되어있는 경우와 반대로 정렬되어있는 경우에만 나타난다. 주어진 n개의 데이터가 이미 오름차순 또는 내림차순으로 정렬되어있을 확률은 미미하다. (n개의 데이터가 모두 다르다고 가정할 때, 2/n!) 따라서 Quick Sort는 보통 $O(nlogn)$time으로 동작한다.

(c) Merge Sort와 Quick Sort를 비교해보았을 때, C_1(the number of key comparisons)의 경우, Merge Sort가 더 작았으나 큰 차이를 보이진 않았다. 하지만, C_2(the number of writing operations on the array to be sorted)의 경우, Merge Sort가 Quick Sort보다 약 4배 정도 큰 값을 보였다. 이는, Quick Sort와 다르게 Merge Sort는 in-place가 불가능하기 때문으로 예상된다.

다시 말해, 키값 비교 횟수는 Merge Sort가 Quick Sort보다 적지만, Merge Sort는 in-place가 불가능해 값을 옮기는 횟수가 Quick Sort보다 매우 크다. 따라서 일반적으로, Quick Sort가 Merge Sort보다 빠르다.