REPORT

Project 1: Comparison-based Sorting Algorithms

Submitted By

1. Abdullah Al Raqibul Islam (ID# 801151189)
2. Jawad Chowdhury, (ID# 801135477)

Code Repository Overview

In this project, we have separate folders for inputs and outputs. All the inputs are being located at the folder **input/** and all the output files are being located at folder **output/.**

Now, each of these input and output folders contain 3 different formatted input and output folders named as 1. **random/** 2. **sorted/** and 3. **rev_sorted/** (Reversely sorted). Each of these folders contain different sample sized files ranging from 5,000 to 50,000.

So for example, file **input/random/in_05k.txt** is an input file containing 5,000 samples in random order and file **output/rev_sorted/out_45k.txt** is an output file containing the sorted output of an input file which was reversely sorted with 45,000 sample inputs.

Other than these, we have the **.cpp** file for Insertion sort (as **insertion_sort.cpp**), Merge sort (as **merge_sort.cpp**), Heapsort (as **heapsort.cpp**), In-place quicksort (as **inplace_quicksort.cpp)** and Modified quicksort (as **modified_quicksort.cpp).**

Also, we have different data generator **.cpp** files such as **data_generator.cpp, data_generator_sorted.cpp, data_generator_rev_sorted.cpp** for generating input data, one .cpp file to compare the solution of the algorithms with the library based sorted output named as **judge_solution.cpp** and different shell scripts to calculate the time needed for each of the sorts such as **calculate_time_random_io.sh, calculate_time_sorted_io.sh** and **calculate_time_rev_sorted_io.sh.**

Please check the **README** file to get source code execution instruction and i/o specification.

Data Structure

In this project we use C++ as our programming language. To implement the sorting algorithms, in most of the cases we used a globally declared array with a max size of 50,005 (as the max size of our input samples is 50,000). We don't use any library function in implementing the algorithms of this project. We prepared several bash scripts to test the execution time of our implemented algorithms.

Complexity Analysis

For **Insertion sort** the running time of different formatted input files are as follows:

1. Sorted inputs (best case): $T(n) = \theta(n)$
2. Reversely sorted inputs (worst case): $T(n) = \theta(n^2)$
3. Random inputs (average case): $T(n) = \theta(n^2)$

For **Merge sort** the running time of different formatted input files are as follows:

1. Sorted inputs: $T(n) = O(nlogn)$
2. Reversely sorted inputs: $T(n) = O(nlogn)$
3. Random inputs: $T(n) = O(nlogn)$

For **Heapsort** the running time of different formatted input files are as follows:

1. Sorted inputs (best case): $T(n) = O(n)$
2. Reversely sorted inputs: $T(n) = O(nlogn)$
3. Random inputs: $T(n) = O(nlogn)$

For **In-place quicksort** the running time of different formatted input files are as follows:

1. Sorted inputs (worst case): $T(n) = O(n^2)$
2. Reversely sorted inputs (worst case): $T(n) = O(n^2)$
3. Random inputs (average case): $T(n) = O(nlogn)$

For **Modified quicksort** the running time of different formatted input files are as follows:

4. Sorted inputs (worst case): $T(n) = O(n^2)$
5. Reversely sorted inputs (worst case): $T(n) = O(n^2)$
6. Random inputs (average case): $T(n) = O(nlogn)$

Results

Due to the **huge scaling difference** of time consumptions between the sorting algorithms for the different cases, we have used the **logarithmic** scale in the **y-axis** for the **input-size vs time** data plotting.

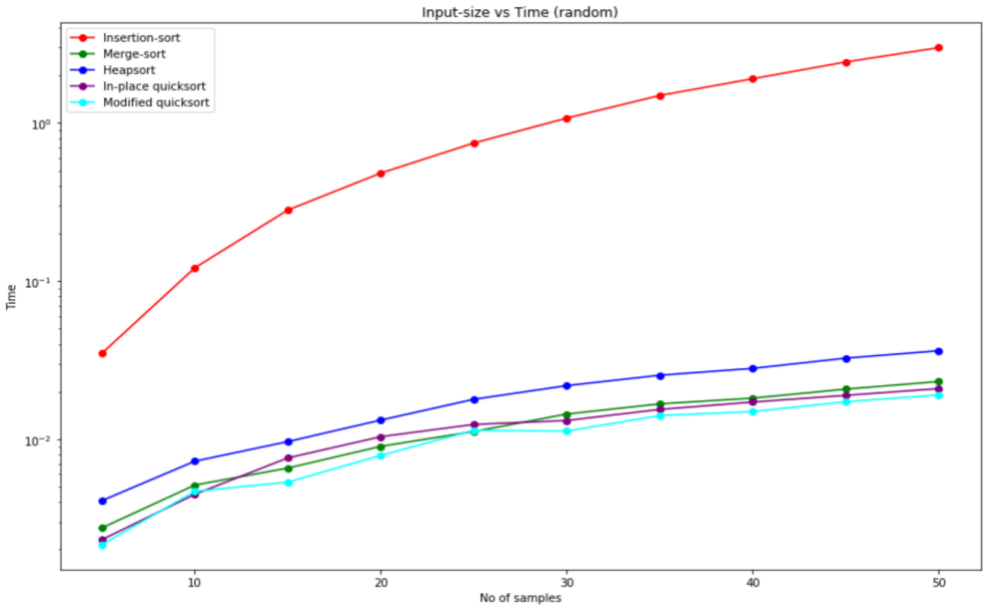

Figure 1: For randomly ordered data (input-size vs time)

In this figure, we can see the time consumption rate for insertion sort is significantly higher (even in the **logarithmic** scale) than all other sorting algorithms (merge sort, heapsort, in-place quicksort and modified quicksort).

As here, the plotting is being done for randomly ordered inputs, the insertion sorting algorithm takes $O(n^2)$ where as most of the other algorithm does it in $O(nlogn)$ times and this is being reflected in the results of this figure.

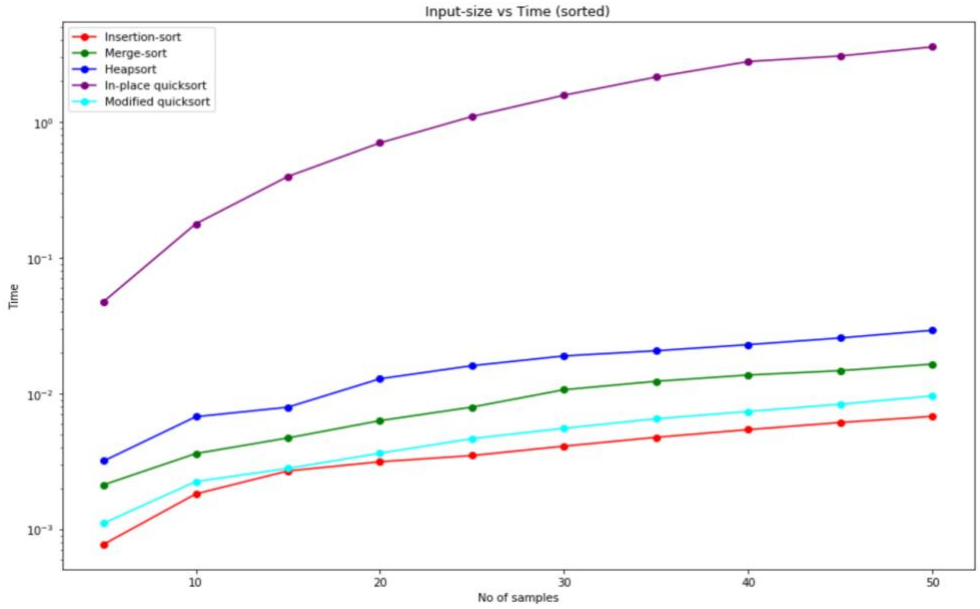

Figure 2: For sorted ordered data (input-size vs time)

In this figure, we can see the time consumption rate for in-place quicksort is much higher than all other sorting algorithms (insertion sort, merge sort, heapsort) as this is the worst case for this algorithm and it takes $O(n^2)$ times.

On the other hand, the sorted input data makes the best case scenario for insertion sorting algorithm as from also the figure we can see the reflection cause in this case the algorithm take $O(n)$ times.

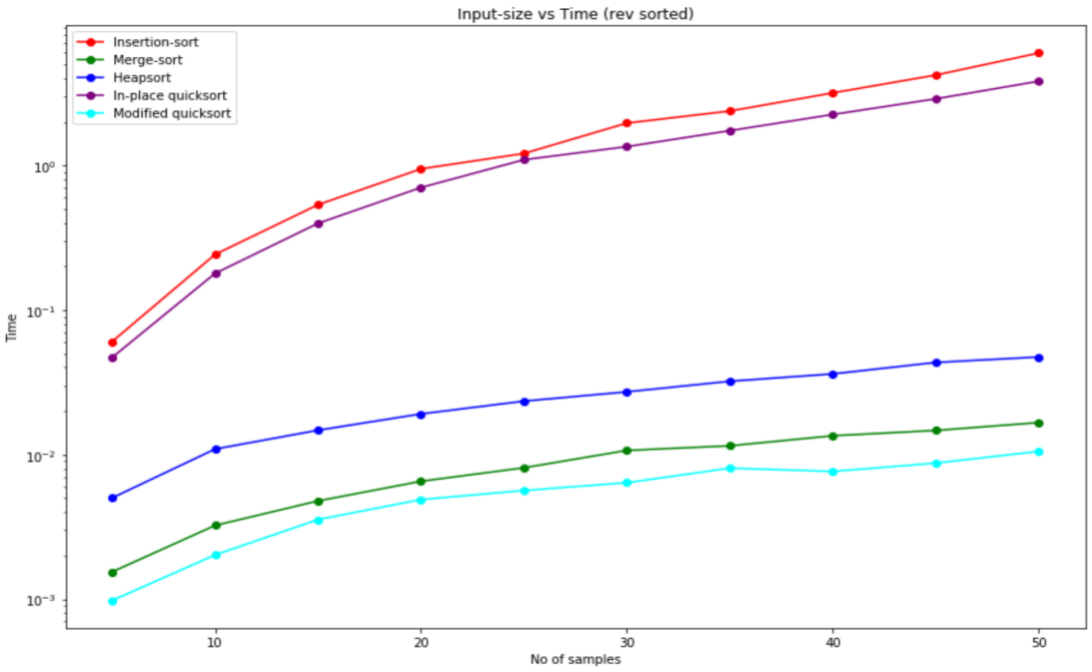

Figure 3: For reversely sorted ordered data (input-size vs time)

In this figure, we can see the time consumption rate for both insertion and in-place quicksort is much higher than all other sorting algorithms (merge sort, heapsort) as this is the worst case for both of these algorithm and they take $O(n^2)$ times.

On the other hand, most of the other algorithm takes much lower time of having a time complexity of $O(nlogn)$.

Code

1.Code for insertion_sort.cpp

```
const int MAX = 50005;

int input_arr[MAX];
int x;


int insertionSort(int n, int start, int end) {
    int i,j,k;
    for(j=start+1; j<=end; j++){
        k=input_arr[j];
        i=j-1;
        while(i>=0 && input_arr[i]>k){
            input_arr[i+1]=input_arr[i];
            i--;
        }
        input_arr[i+1]=k;
    }
    return 0;
}

int main() {
    //freopen("in.txt", "r", stdin);
    //freopen("out.txt", "w", stdout);

    int i, j, k;
    int test, t = 0, kase = 0;
    //
    int n;
    scanf("%d", &x);
    n = x;
    for(i=0; i<n; i+=1) {
        scanf("%d", &input_arr[i]);
    }
    double st = clock();
```

```
    int start =0;
    int end = n-1;
    insertionSort(n, start, end);
    for(i=0; i<n; i+=1) {
        printf("%d\n", input_arr[i]);
    }
    cerr << (clock() - st) / CLOCKS_PER_SEC << endl;
    //
    return 0;
}
```

2.Code for merge_sort.cpp

```
const int MAX = 50005;

int input_arr[MAX];
int x;

int merge(int n, int start, int mid, int end){
    int n1= mid - start +1;
    int n2= end-mid;
    int leftArray[n1];
    int rightArray[n2];
    int i,j,k;
    for(i=0; i<n1; i++){
        leftArray[i]=input_arr[start+i];
    }
    for(j=0; j<n2; j++){
        rightArray[j]=input_arr[mid+1+j];
    }
    i=0, j=0, k=start;
    while(i<n1 && j<n2){
        if(leftArray[i]<=rightArray[j]){
            input_arr[k++]=leftArray[i++];
        }else{
            input_arr[k++]=rightArray[j++];
        }
```

```
    }
    while(i<n1){
        input_arr[k++]=leftArray[i++];
    }
    while(j<n2){
        input_arr[k++]=rightArray[j++];
    }

    return 0;
}

int mergeSort(int n, int start, int end) {
    if (start < end){
        int mid = (end+start)/2;
        mergeSort(n, start, mid);
        mergeSort(n, mid+1, end);
        merge(n, start, mid, end);
    }
    return 0;
}

int main() {
    //freopen("in.txt", "r", stdin);
    //freopen("out.txt", "w", stdout);

    int i, j, k;
    int test, t = 0, kase = 0;
    //
    int n;
    scanf("%d", &x);
    n = x;
    for(i=0; i<n; i+=1) {
        scanf("%d", &input_arr[i]);
    }
    double st = clock();
    int start =0;
    int end = n-1;
```

```
    mergeSort(n, start, end);
    for(i=0; i<n; i+=1) {
        printf("%d\n", input_arr[i]);
    }
    cerr << (clock() - st) / CLOCKS_PER_SEC << endl;
    //
    return 0;
}
```

3.Code for heapsort.cpp

```
const int MAX = 50005;

int heap[MAX];
int input_arr[MAX];
int n;

void heap_insert(int key, int pos) {
    heap[pos] = key;
    int i = pos;
    while(i>1 && heap[i/2] > heap[i]) {
        swap(heap[i], heap[i/2]);
        i = i / 2;
    }
}

int heap_remove_min() {
    int j, i = 1, tmp = heap[1];
    heap[1] = heap[n];
    n -= 1;

    while(i < n) {
        if(2*i+1 <= n) {    //heap node has 2 child
            if(heap[i] <= heap[2*i] && heap[i] <=
heap[2*i+1]) {
                return tmp;
            }
```

```
            else {
                if(heap[2*i] < heap[2*i+1]) j = 2*i;
                else j = 2*i+1;

                swap(heap[i], heap[j]);
                i = j;
            }
        }
        else {  //heap node has 0 or 1 child
            if(2*i <= n && heap[i] > heap[2*i]) {
                swap(heap[i], heap[2*i]);
            }
            return tmp;
        }
    }
    return tmp;
}

int main() {
    //freopen("input/in_50k.txt", "r", stdin);
    //freopen("out.txt", "w", stdout);

    int i, j, k;
    int test, t = 0, kase=0;
    int key, _n;

    scanf("%d", &n);
    _n = n;
    //printf("%d\n", n);
    for(i=1; i<=_n; i+=1) {
        scanf("%d", &input_arr[i]);
    }

    double st=clock();

    for(i=1; i<=_n; i+=1) {
        heap_insert(input_arr[i], i);
```

```
    }

    for(i=0; i<_n; i+=1) {
        //if(i) printf(" ");
        printf("%d\n", heap_remove_min());
    }
    //printf("\n");
    cerr << (clock()-st)/CLOCKS_PER_SEC << endl;

    return 0;
}
```

4.Code for inplace_quicksort.cpp

```
const int MAX = 50005;

int n;
int input_arr[MAX];

int get_pivot(int left, int right) {
    return input_arr[left];
}

void inplace_quick_sort(int left, int right) {
    //base case
    if(left >= right) return;

    int pivot = get_pivot(left, right);
    int i = left, j = right;

    while(true) {
        while(input_arr[i] < pivot) i+=1;
        while(pivot < input_arr[j]) j -= 1;
        if(i < j) {
            swap(input_arr[i], input_arr[j]);
            i += 1;
            j -= 1;
```

```
        }
        else break;
    }
    inplace_quick_sort(left, i-1);
    inplace_quick_sort(j+1, right);
}

int main() {
   //freopen("input/random/in_50k.txt", "r", stdin);
   //freopen("out.txt", "w", stdout);

   int i, j, k;
   int test, t = 0, kase=0;

   scanf("%d", &n);
    for(i=0; i<n; i+=1) {
        scanf("%d", &input_arr[i]);
    }

    double st=clock();

    inplace_quick_sort(0, n-1);

    for(i=0; i<n; i+=1) {
        printf("%d\n", input_arr[i]);
    }

    cerr << (clock()-st)/CLOCKS_PER_SEC << endl;

   return 0;
}
```

5.Code for modified_quicksort.cpp

```
const int MAX = 50005;

int n;
```

```
int input_arr[MAX];

int median_of_three(int a, int b, int c) {
    if ((b <= a && a <= c) || (c <= a && a <= b))
return a;
    if ((a <= b && b <= c) || (c <= b && b <= a))
return b;
    return c;
}

int get_pivot(int left, int right) {
    int mid = (left + right) / 2;
    return median_of_three(input_arr[left],
input_arr[mid], input_arr[right]);
}

int insertion_sort(int start, int end) {
    int i, j, k;
    for (j = start + 1; j <= end; j++) {
        k = input_arr[j];
        i = j - 1;
        while (i >= 0 && input_arr[i] > k) {
            input_arr[i + 1] = input_arr[i];
            i--;
        }
        input_arr[i + 1] = k;
    }
    return 0;
}

void inplace_quick_sort(int left, int right) {
    //base case
    if (left + 10 <= right) {
        int pivot = get_pivot(left, right);
        int i = left, j = right;

        while (true) {
```

```
            while (input_arr[i] < pivot) i += 1;
            while (pivot < input_arr[j]) j -= 1;
            if (i < j) {
                swap(input_arr[i], input_arr[j]);
                i += 1;
                j -= 1;
            } else break;
        }
        inplace_quick_sort(left, i - 1);
        inplace_quick_sort(j + 1, right);
    } else {
        insertion_sort(left, right);
    }
}

int main() {
    //freopen("input/random/in_40k.txt", "r", stdin);
    //freopen("out.txt", "w", stdout);

    int i, j, k;
    int test, t = 0, kase = 0;

    scanf("%d", &n);
    for (i = 0; i < n; i += 1) {
        scanf("%d", &input_arr[i]);
    }

    double st = clock();

    inplace_quick_sort(0, n - 1);

    for (i = 0; i < n; i += 1) {
        printf("%d\n", input_arr[i]);
    }

    cerr << (clock() - st) / CLOCKS_PER_SEC << endl;
```

```
    return 0;
}
```