Project 1 Report

Joey Parker

University of North Carolina at Charlotte

ITCS 6114-81

Dr. Dewan Ahmed

July 7, 2024

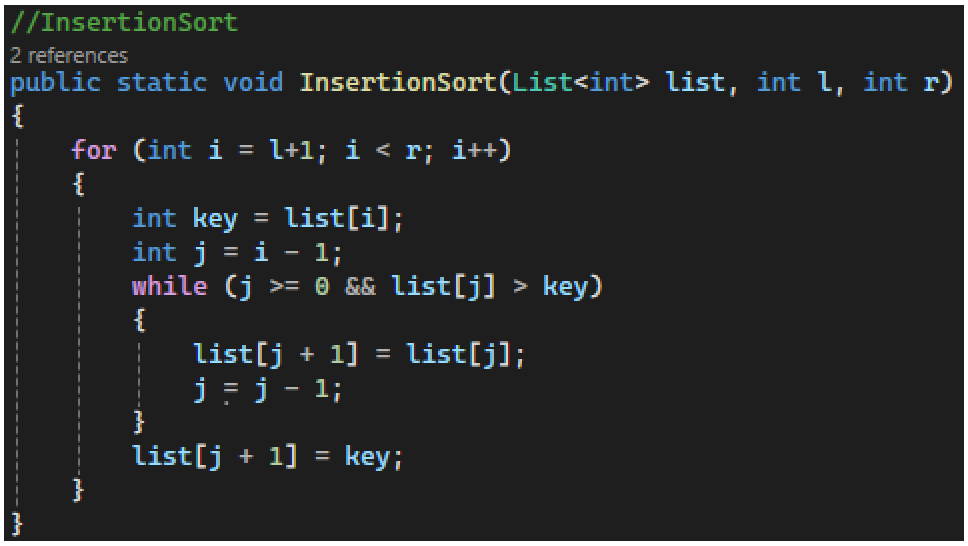
Project GitHub Repository: https://github.com/Ninjajkl/ITCS-6114-Project-1/

Table of Contents

1 Insertion Sort

Insertion Sort is a very simple sorting algorithm that builds the final sorted array one item at a time. It iteratively places each element in its correct position relative to previously sorted elements. In terms of best-case analysis, Insertion Sort has a **O(n)** time complexity, as each element would only be checked once. In terms of worst-case analysis, Insertion Sort has a **O(n^2)** time complexity, as each element would have to be checked against every other element n-1 times. For my implementation of Insertion Sort, I used an in-place C# List for my sequence.

Here is my code for Insertion Sort:

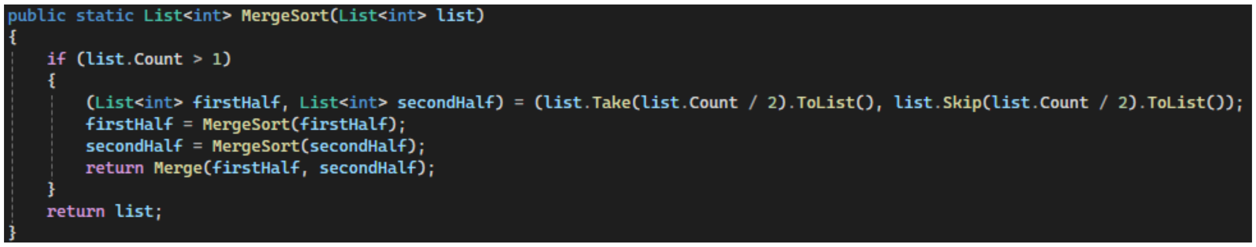


```
//InsertionSort
2 references
public static void InsertionSort(List<int> list, int l, int r)
{
    for (int i = l+1; i < r; i++)
    {
        int key = list[i];
        int j = i - 1;
        while (j >= 0 && list[j] > key)
        {
            list[j + 1] = list[j];
            j = j - 1;
        }
        list[j + 1] = key;
    }
}
```

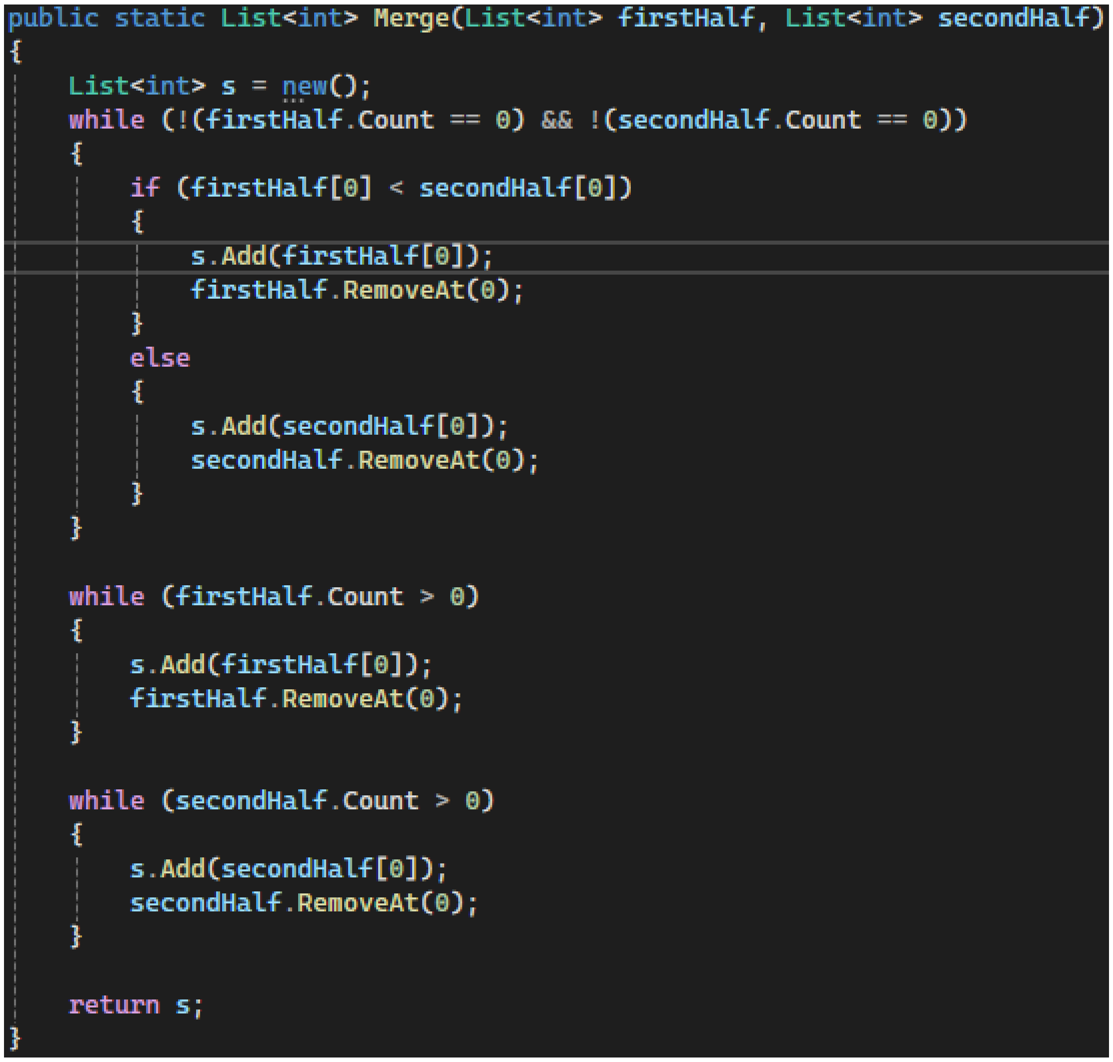
2 Merge Sort

Merge Sort is a divide-and-conquer sorting algorithm that recursively divides an array into halves, sorts each half, and merges them back in sorted order. In terms of best-case analysis, Merge Sort has a **O(n log n)** time complexity, which is the same as the worst-case analysis, **O(n log n)**. This is because Merge Sort makes the same number of comparisons regardless of input order. The height *h* of the merge-sort tree has **O(log n)** time complexity, as at each recursive call we divide in half the sequence. The overall amount of work done at the nodes of depth *i* is **O(n)**. This is because we partition and merge 2^i sequences of size n/2, therefore making 2^{i+1} recursive calls. Thus, the total running time of merge-sort is **O(n log n)**. For my implementation of Merge Sort, I used List<int> as my sequences. Notably, the algorithm is not performed in-place, and it creates many sub-lists to sort.

Here is my code for Merge Sort:



```
public static List<int> MergeSort(List<int> list)
{
    if (list.Count > 1)
    {
        (List<int> firstHalf, List<int> secondHalf) = (list.Take(list.Count / 2).ToList(), list.Skip(list.Count / 2).ToList());
        firstHalf = MergeSort(firstHalf);
        secondHalf = MergeSort(secondHalf);
        return Merge(firstHalf, secondHalf);
    }
    return list;
}
```

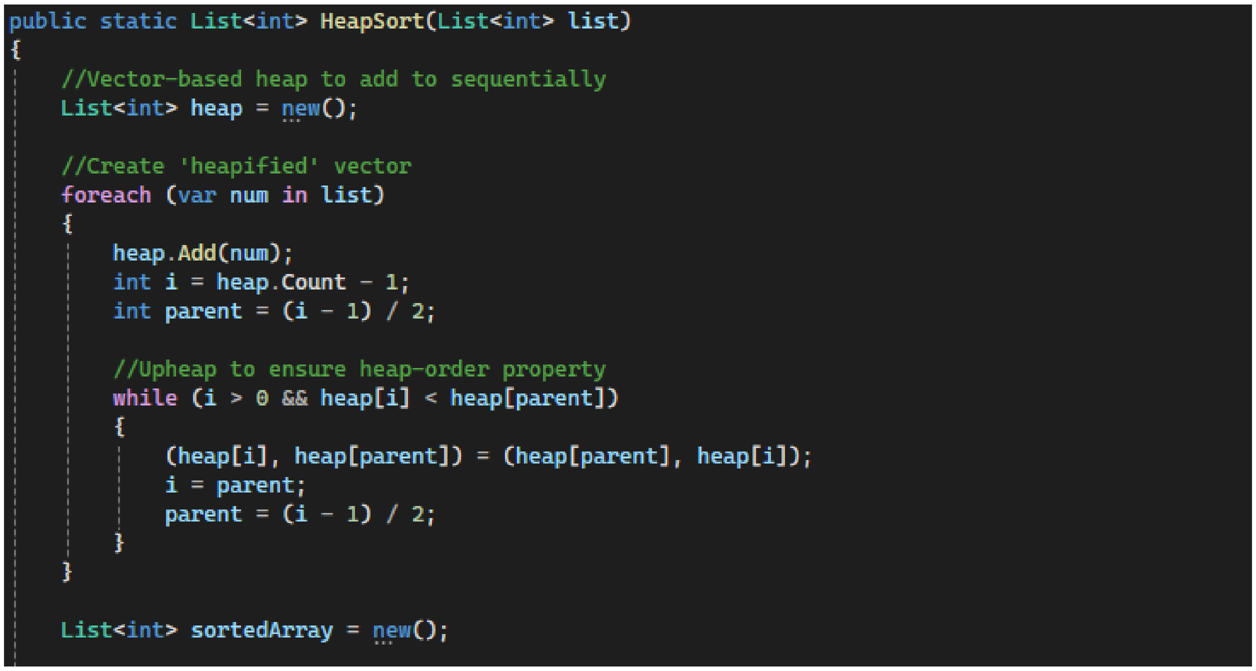


```
public static List<int> Merge(List<int> firstHalf, List<int> secondHalf)
{
    List<int> s = new();
    while (!(firstHalf.Count == 0) && !(secondHalf.Count == 0))
    {
        if (firstHalf[0] < secondHalf[0])
        {
            s.Add(firstHalf[0]);
            firstHalf.RemoveAt(0);
        }
        else
        {
            s.Add(secondHalf[0]);
            secondHalf.RemoveAt(0);
        }
    }

    while (firstHalf.Count > 0)
    {
        s.Add(firstHalf[0]);
        firstHalf.RemoveAt(0);
    }

    while (secondHalf.Count > 0)
    {
        s.Add(secondHalf[0]);
        secondHalf.RemoveAt(0);
    }

    return s;
}
```

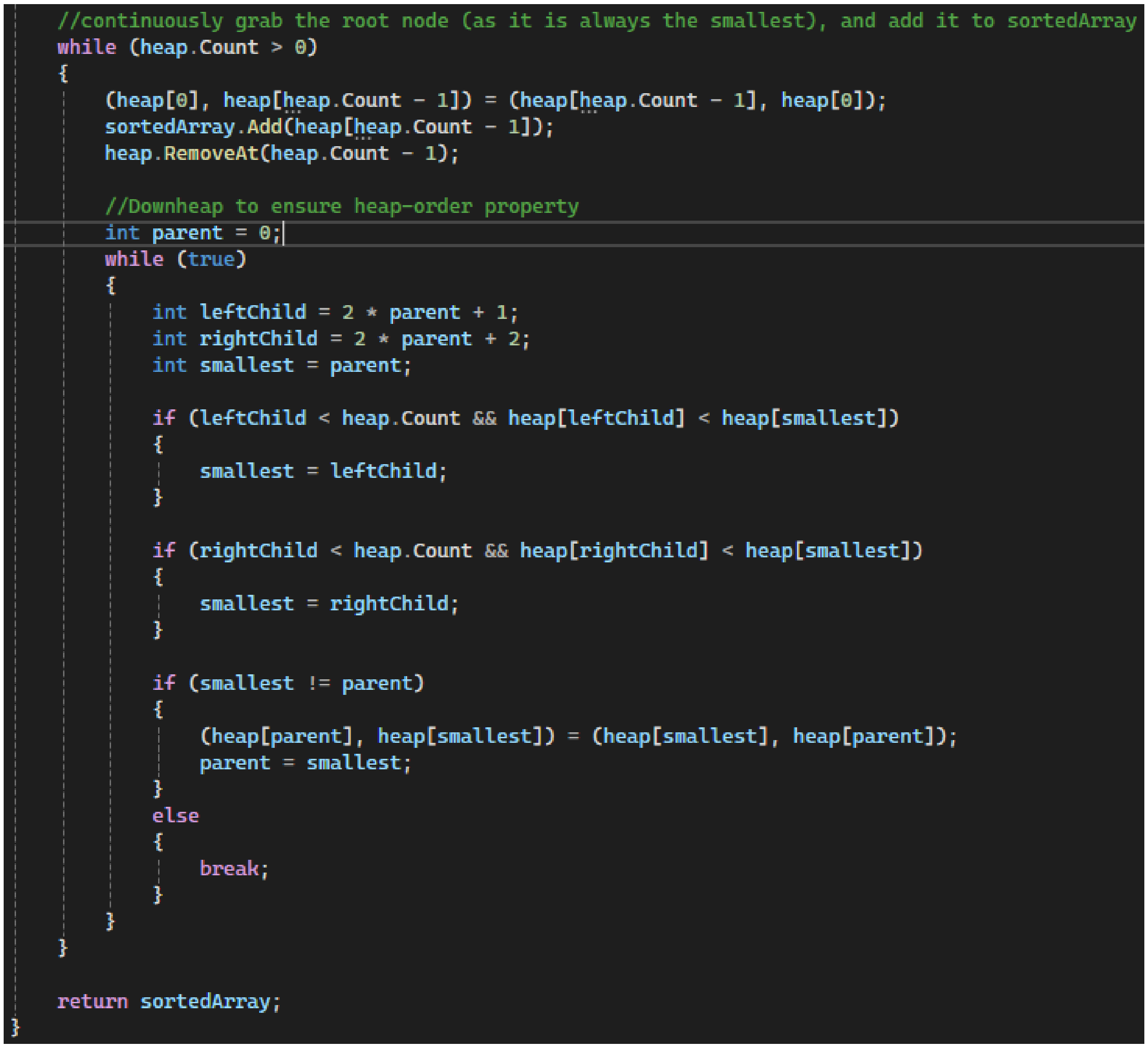
3 Heap Sort

Heap Sort is a comparison-based sorting technique that uses a binary heap data structure to repeatedly extract the maximum (for max heap) or minimum (for min heap) element and rebuild the heap. It sorts the array in place (as it is being inserted into/ removed from). This implementation used a min heap (as I wanted the root to always be the minimum). In terms of worst-case analysis, Heap Sort has a **O(n log n)** time complexity. This is because it you need to insert each element once each into the heap, for **O(n)**. In the worst case, every item would be distinct, requiring the downheap to be ran for every insert, which has a time complexity of **O(log n)**. Inserting n distinct items in the worst case would cause the downheap to run every time, causing a total **O(n log n)** time complexity. In terms of best-case analysis, Heap Sort has a **O(n)** time complexity. This is due to the Heap-order property never being violated, which means the downheap is never ran. For my implementation of Heap Sort, I used three List<int> for my vectors. The first was the input vectors, which inserted into the second List<int>, the actual hash. The last was the sortedArray, which had the always-minimum root node added to it to create the sorted sequence. Notably, the algorithm is **not** performed in-place, and is vector-based.

Here is my code for Heap Sort:



```
public static List<int> HeapSort(List<int> list)
{
    //Vector-based heap to add to sequentially
    List<int> heap = new();

    //Create 'heapified' vector
    foreach (var num in list)
    {
        heap.Add(num);
        int i = heap.Count - 1;
        int parent = (i - 1) / 2;

        //Upheap to ensure heap-order property
        while (i > 0 && heap[i] < heap[parent])
        {
            (heap[i], heap[parent]) = (heap[parent], heap[i]);
            i = parent;
            parent = (i - 1) / 2;
        }
    }

    List<int> sortedArray = new();
```

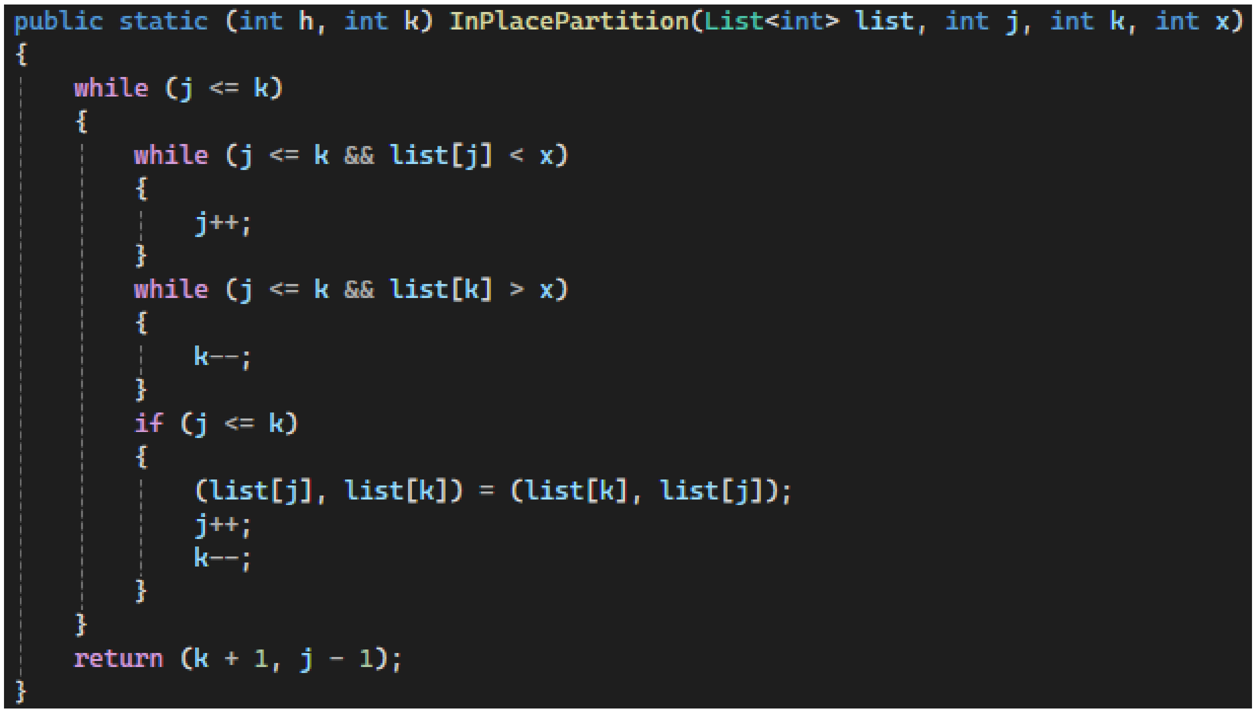


```
    //continuously grab the root node (as it is always the smallest), and add it to sortedArray
    while (heap.Count > 0)
    {
        (heap[0], heap[heap.Count - 1]) = (heap[heap.Count - 1], heap[0]);
        sortedArray.Add(heap[heap.Count - 1]);
        heap.RemoveAt(heap.Count - 1);

        //Downheap to ensure heap-order property
        int parent = 0;
        while (true)
        {
            int leftChild = 2 * parent + 1;
            int rightChild = 2 * parent + 2;
            int smallest = parent;

            if (leftChild < heap.Count && heap[leftChild] < heap[smallest])
            {
                smallest = leftChild;
            }

            if (rightChild < heap.Count && heap[rightChild] < heap[smallest])
            {
                smallest = rightChild;
            }

            if (smallest != parent)
            {
                (heap[parent], heap[smallest]) = (heap[smallest], heap[parent]);
                parent = smallest;
            }
            else
            {
                break;
            }
        }
    }

    return sortedArray;
}
```

4 Quick Sort

Quick Sort is a divide-and-conquer algorithm that selects a pivot element and partitions the array into two sub-arrays: elements less than the pivot and elements greater than the pivot. It then recursively sorts these sub-arrays. In terms of best-case analysis, Quick Sort has a **O($nlog_2n$)** time complexity. This would be when the array is cut in half each iteration, causing the depth of the recursion to be **O(log_2n)**. At each level of the recursion, all the partitions at that level do work in **O(n)**. This causes the best case to be **O($nlog_2n$)**. In terms of worst-case analysis, Quick Sort has a **O(n^2)** time complexity. This would occur when the pivot is the unique minimum or maximum element. The running time would be proportional to the sum $n + (n - 1) + \cdots + 2 + 1$, causing the **O($n^2$)** time complexity. Both my In-Place and Modified Quick Sort algorithms used the same In-Place Partition code.

Here is the In-Place Partition Code:

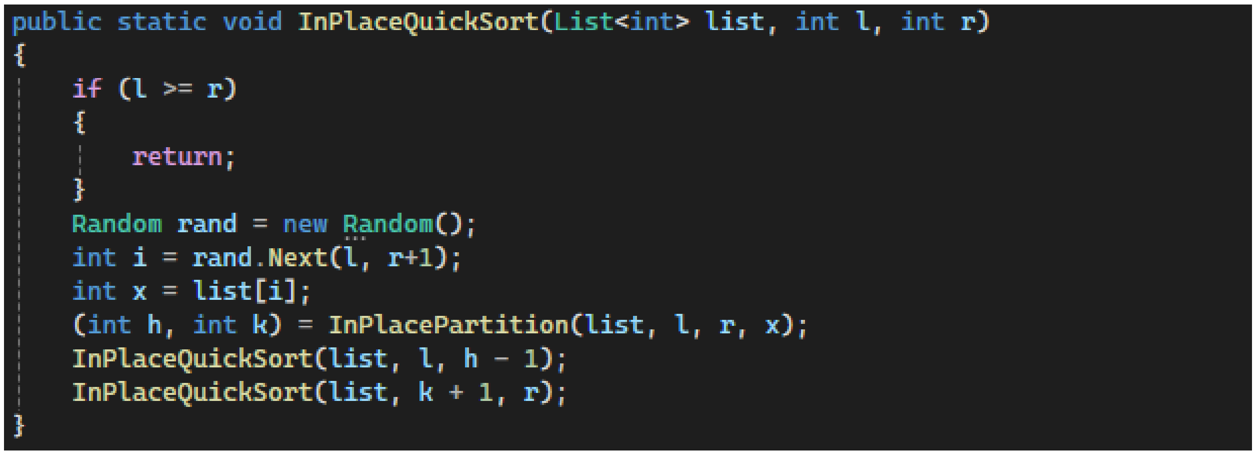


```
public static (int h, int k) InPlacePartition(List<int> list, int j, int k, int x)
{
    while (j <= k)
    {
        while (j <= k && list[j] < x)
        {
            j++;
        }
        while (j <= k && list[k] > x)
        {
            k--;
        }
        if (j <= k)
        {
            (list[j], list[k]) = (list[k], list[j]);
            j++;
            k--;
        }
    }
    return (k + 1, j - 1);
}
```

4.1 In-Place Quick Sort

For my In-Place implementation of Quick Sort, I used a List<int> for my sequence. For my pivot point, I opted to choose a random item.

Here is my Code for In-Place Quick Sort:

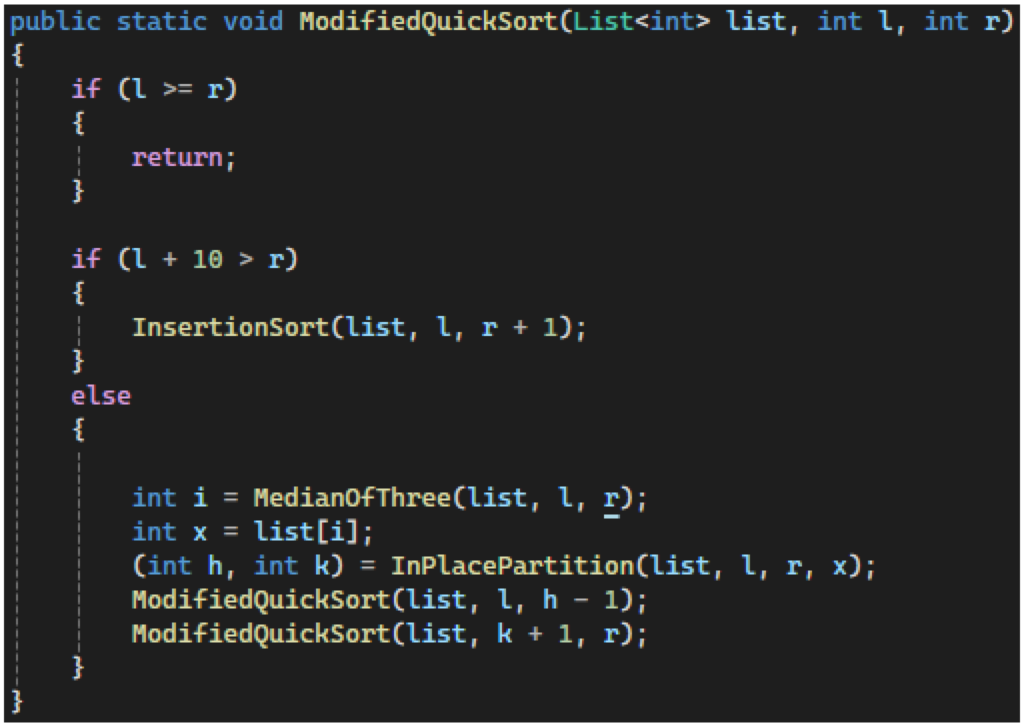


```
public static void InPlaceQuickSort(List<int> list, int l, int r)
{
    if (l >= r)
    {
        return;
    }
    Random rand = new Random();
    int i = rand.Next(l, r+1);
    int x = list[i];
    (int h, int k) = InPlacePartition(list, l, r, x);
    InPlaceQuickSort(list, l, h - 1);
    InPlaceQuickSort(list, k + 1, r);
}
```

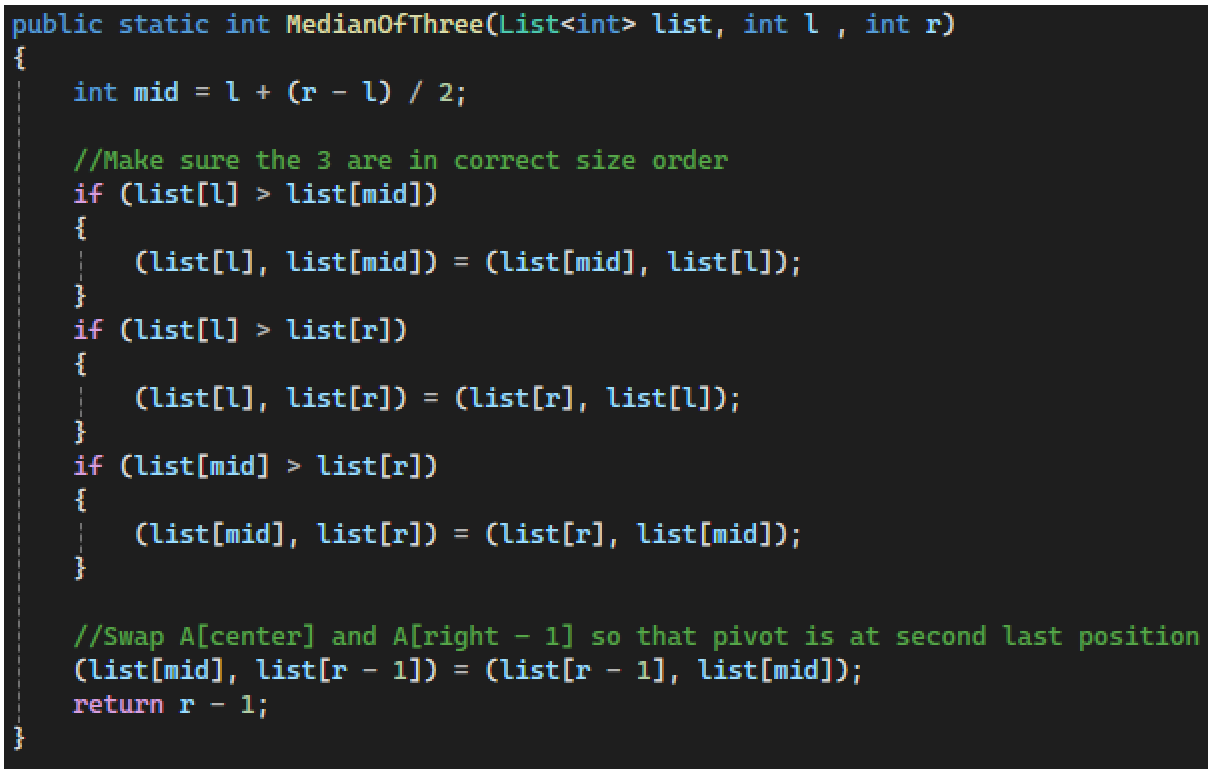
4.2 Modified Quick Sort

For my modified implementation of Quick Sort, I used a List<int> for my sequence. For my pivot point, I used median-of-three. For sub-problems of size less than 10, I used my (In-Place) Insertion Sort code from above. Notably, this modified Quick Sort is also implemented in-place.

Here is my Code for Modified Quick Sort:



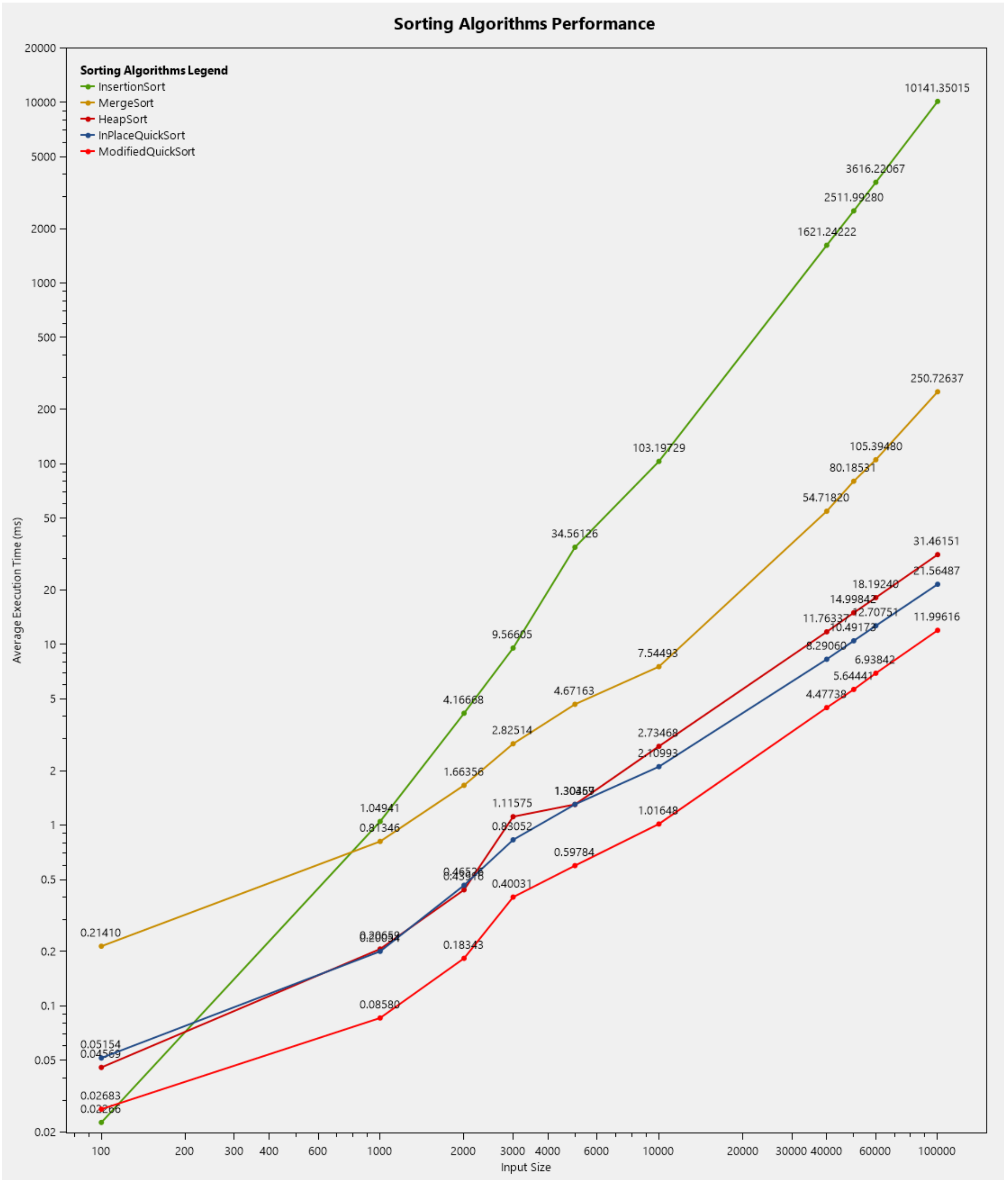
```
public static void ModifiedQuickSort(List<int> list, int l, int r)
{
    if (l >= r)
    {
        return;
    }

    if (l + 10 > r)
    {
        InsertionSort(list, l, r + 1);
    }
    else
    {

        int i = MedianOfThree(list, l, r);
        int x = list[i];
        (int h, int k) = InPlacePartition(list, l, r, x);
        ModifiedQuickSort(list, l, h - 1);
        ModifiedQuickSort(list, k + 1, r);
    }
}
```



```
public static int MedianOfThree(List<int> list, int l , int r)
{
    int mid = l + (r - l) / 2;

    //Make sure the 3 are in correct size order
    if (list[l] > list[mid])
    {
        (list[l], list[mid]) = (list[mid], list[l]);
    }
    if (list[l] > list[r])
    {
        (list[l], list[r]) = (list[r], list[l]);
    }
    if (list[mid] > list[r])
    {
        (list[mid], list[r]) = (list[r], list[mid]);
    }

    //Swap A[center] and A[right - 1] so that pivot is at second last position
    (list[mid], list[r - 1]) = (list[r - 1], list[mid]);
    return r - 1;
}
```

5 Simulation

For my simulation, I ran each algorithm for inputs of size 100, 1000, 2000, 3000, 5000, 10000, 40000, 50000, 60000, and 100000. Each input has values between 0 and n. For each input size, I generated 10 different datasets. I then ran each dataset on every Sorting Algorithm, then averaged them. This means 500 sequences of various sizes were sorted. The resulting graph of this data is shown below:



5.1 Results Analysis

From the graph, it is clear that Modified Quick Sort is the best algorithm for nearly all input lengths, except for tiny inputs. Insertion Sort, on the complete flip side, performs by far the best for tiny inputs, and just terrible for anything else. Insertion Sort took on average 10,141ms (10.14s) for inputs of size 100,000. Clearly, Insertion sort with its $\mathbf{O(n^2)}$ worst-case time complexity should not be used for anything larger than 100 elements. Merge Sort performed the second-worst. It took the second-longest for nearly every task, despite having the same worst-case time analysis as heap-sort, **O(n log n)**. I believe this is caused by the sheer amount of overhead the algorithm requires as it continuously makes new list<int>. Heap Sort and In-Place Quick Sort performed almost identically at lower input sizes. In-Place Quick Sort does out-perform Heap Sort starting at inputs of size 10000, with the gap between their performances increasing quickly. Even with how fast In-Place Quick Sort is, Modified Quick Sort is faster. I am not sure whether the Median Of Three or sub-problem Insertion Sort helped contribute to this more, but clearly, they significantly improve the sorting time. At input size of 100,000, Modified Quick Sort is twice as fast as In-Place Quick Sort, thrice as fast as Heap Sort, 250 times as fast as Marge Sort, and 1014 times as fast as Insertion Sort. Modified Quick Sort is simply the best for general sorting as it performs nearly the best at small inputs, and by far the best at all other input sizes.

5.2 Special Cases

For the special cases, I ran 10 sequences of data of input size 50,000 through each sorting algorithm and averaged the results. The data for the sorted sequences is different than the data for the reverse sorted sequences.

5.2.1 Sorted Input

Insertion Sort	Merge Sort	Heap Sort	In-Place Quick Sort	Modified Quick Sort
0.23727	71.72271	12.1302	6.9436	2.08362

From the table above, it is clear that Insertion Sort is the best for sorted inputs. This is because a sorted input is Insertion Sort's best-case scenario with a time complexity of **O(n)**. Insertion sort works by checking if each value is higher than the previous continuously. If it never finds a decreasing value, it will check each input once then finish. Merge Sort has by far the worst time complexity on the sorted input. This is because Merge Sort makes the same number of comparisons regardless of input order. Heap Sort does the second worst, followed by In-Place Quick Sort, then Modified Quick Sort.

5.2.2 Reverse Sorted Input

Insertion Sort	Merge Sort	Heap Sort	In-Place Quick Sort	Modified Quick Sort
5095.17714	72.88355	17.40573	7.01021	2.25601

From the table above, it is clear that Insertion Sort is the worst for reverse-sorted inputs. In fact, reverse-sorted input is the worst-case scenario for Insertion Sort. This is because it will cause each element in the sequence to be checked by every single other element in the sequence by n-1 times. If we have n items in this sequence, this means the time complexity is $\mathbf{O(n^2)}$. This shows in the results, Insertion Sort is nearly 70 times slower than the next slowest algorithm, Merge Sort. Comparing this table with the Sorted Input table, every algorithm but Insertion Sort and Heap Sort perform nearly identically. Insertion sort is 21,229 times slower at sorting reverse-sorted inputs than sorted inputs. Heap sort is only 1.4 times slower at reverse-sorted inputs than sorted inputs. The best reverse-sorted input algorithm is Modified Quick Sort at 2.26 ms.

5.3 Takeaway

In general use cases, Merge-sort is the best sorting algorithm. It performs extremely well in small input sizes, the best at medium to huge input sizes, extremely well with sorted inputs, and the best with reverse-sorted inputs. The only time to not choose Merge-sort is if you either are only sorting tiny inputs or already sorted inputs, in which case Insertion Sort is better.