Sorting

Laasyasree Desu

Fall 2023

Introduction

In this assignment the objective was to implement and compare different sorting algorithms. The algorithms include selection sort, insertion sort, merge sort and an improved merge sort with an extra array. The algorithms were then benchmarked to compare the the time taken.

Selection sort

The first algorithm implemented was selection sort. This algorithm was simple to implement. It starts with a pointer keeping track of the value at the first index. A second pointer then traverses the entire array to find the smallest value. Once complete the minimum value then switches places with the first value, and the first pointer is incremented (points to the second index now). This continues until the array is sorted. This was implemented using two for loops (nested), one to traverse the array, and one to keep track of the first pointer. To swap the elements a temporary variable was created. An example of the code can be seen below:

```
public static void selection(int arr[]) {
        int length = arr.length;
        for (int i = 0; i < length - 1; i++) {
            int candidate = i;
            for (int j = i + 1; j < length; j++) {
                if (arr[j] < arr[candidate])
                    candidate = j;
                    }
            int temp = arr[candidate];
            arr[candidate] = arr[i];
            arr[i] = temp;
            }
        return;
    }
```

Benchmark

Array size	Time taken (ns)
1000	215
2000	917
4000	3391
8000	11058
16000	45930

Table 1: Average time taken to sort an array of various lengths using selection sort

The table above shows the time taken taken to sort an array using selection sort. For this algorithm, to find minimum value and sort, the entire array must be traversed, which can be seen in the for loop. This gives time complexity O(n). Since there are two for loops (nested), the time complexity can be concluded to be O(n^2). This can also be seen (ref. table 1) in how the time grows exponentially in relation to the array size.

Insertion sort

The second algorithm implemented was insertion sort. For this algorithm it can be thought that the array is separated into two parts, sorted and unsorted, where elements from the unsorted are put into the sorted (in the right place). Similar to to above the array is traversed from the start but this time if the predecessor is less than the current element, they are swapped. The swapping continues to happen until there is no larger element being compared. This algorithm was also implemented with two for loops, this time the second for loop worked backwards from the current index. An example of the code can be seen below:

```
public static void insertion(int arr[]){
    for(int i=1; i<arr.length; i++){
        for (j=i; j > 0 && arr[j-1] > arr[i]; j--) {

            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
```

Benchmark

Array size	Time taken (ns)
1000	64
2000	264
4000	1031
8000	3854
16000	14202

Table 2: Average time taken to sort an array of various lengths using insertion sort

The table above shows the time taken taken to sort an array using insertion sort. Similar to the selection sort, to find minimum value and sort the entire array is still traversed, which can also be seen in the for loop giving the time complexity O(n). Since there are two for loops (nested), the time complexity can be concluded to be O(n^2) which can also be seen (ref. table 2) (the time growing exponentially). This algorithm is also observed to be relatively more time efficient - the selection sort algorithm takes approximately 3 to 4 times longer for the same array sizes.

Merge sort

The third algorithm implemented was merge sort. This algorithm can be thought to use the ”divide and conquer” strategy. The algorithm starts by creating an extra array of the same size to store temporary values. Then it divides the original array into two parts. The two parts are sorted separately and placed into different arrays. These arrays are then merged to result in a sorted array. This algorithm was implemented by creating three different methods. The first method created the temporary array. The second method divided the array and the two parts are sorted through recursive calls (continuously split in half). The third method deals with merging the arrays. This is done by comparing the values in both arrays, if the first element in the first array is smaller than the first element in the second array, that element is placed in the original array and vice versa. An example of the code can be seen below. The first section of code shows the recursive part of the code, where the array is split continuously. The second section of code shows how the comparison and merging to for the final sorted array.

```
private static void sort(int[] org, int[] aux, int lo, int hi) {
if (lo != hi) {
int mid = (lo + hi)/2;
    int mid = (lo + hi)/2;
    sort(org, aux, lo, mid)
    sort(org, aux, mid +1, hi)
    merge(org, aux, lo, mid, hi);
}
```

```
for (int k = lo; k <= hi; k++) {
    if (i > mid) {
    org[k] = aux[j++];
                }
    else if (j > hi) {
    org[k] = aux[i++];
                }
    else if (aux[i] < aux[j]) {
    org[k] = aux[i++];
                }
    else {org[k] = aux[j++];
            }
        }
```

Benchmark

Array size	Time taken (ns)
1000	77
2000	115
4000	311
8000	704
16000	1152

Table 3: Average time taken to find duplicates using pointers in sorted array.

The time complexity for this algorithm can be determined by the different parts of the algorithm. The first part deals with repeatedly dividing the algorithm by two. This gives the expression log($_2$n), resulting in the time complexity

O(log (n)). The second part deals with the merging, which depends on the size pf the array, giving the time complexity O(n). This results in the algorithms time complexity being O(n log(n)). This can also be seen from the results, as the the time taken to sort was significantly lower than the other two algorithms, especially for larger array sizes (as there is not an exponential growth).