Sorting an array in C

Ying Pei Lin

Fall 2024

Selection Sort

From the start of the array, the algorithm searches for the smallest element in the unsorted part of the array(the right side of the current element) and swaps it with the current element, then moves to the next element. If there are n elements in the array, the algorithm will perform $(n - i)$ comparisons for the ith element, and the total number of comparisons is:

$$n + (n - 1) + (n - 2) + \ldots + 1 = \frac{n(n + 1)}{2} \tag{1}$$

Therefore, the time complexity of the selection sort is $O(n^2)$. The following code snippet shows the implementation of the selection sort algorithm and the result of sorting array with different sizes is shown in Figure 1.

```
void selection_sort(int *array, int length) {
  for (int i = 0; i < length - 1; i++) {
    int min_index = i;
    for (int j = i + 1; j < length; j++) {
      if (array[j] < array[min_index]) {
        min_index = j;
      }
    }
    int temp = array[min_index];
    array[min_index] = array[i];
    array[i] = temp;
  }
}
```

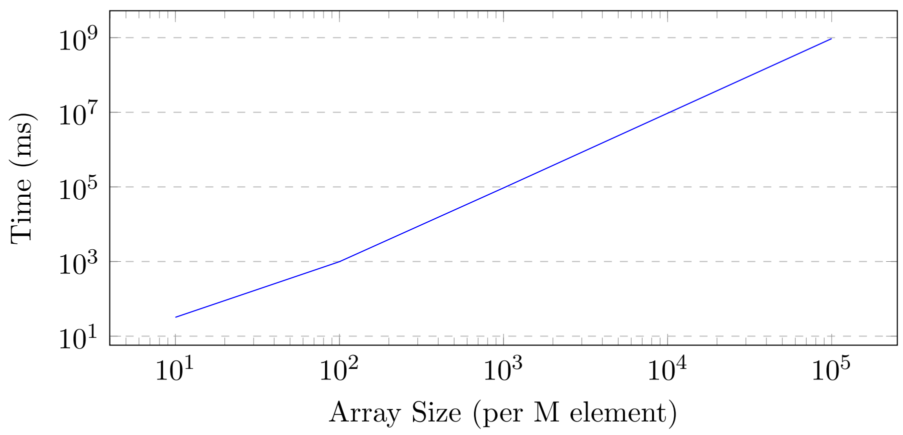

Figure 1: Selection Sort with different array sizes

Insertion Sort

The insertion sort algorithm works by finding the correct position at the left side of the current element. If there are n elements in the array, the algorithm will perform $(n - i)$ comparisons for the ith element, and the total number of comparisons is the same as the selection sort algorithm, which is $\frac{n(n+1)}{2}$. Therefore, the time complexity of the insertion sort is also $O(n^2)$.

The following code snippet implements the insertion sort algorithm and the result is shown in Figure 2.

```
void insertion_sort(int *array, int length) {
  int i, j, key;
  for (i = 1; i < length; i++) {
    key = array[i];
    j = i - 1;
    while (j >= 0 && array[j] > key) {
      array[j + 1] = array[j];
      j = j - 1;
    }
    array[j + 1] = key;
  }
}
```

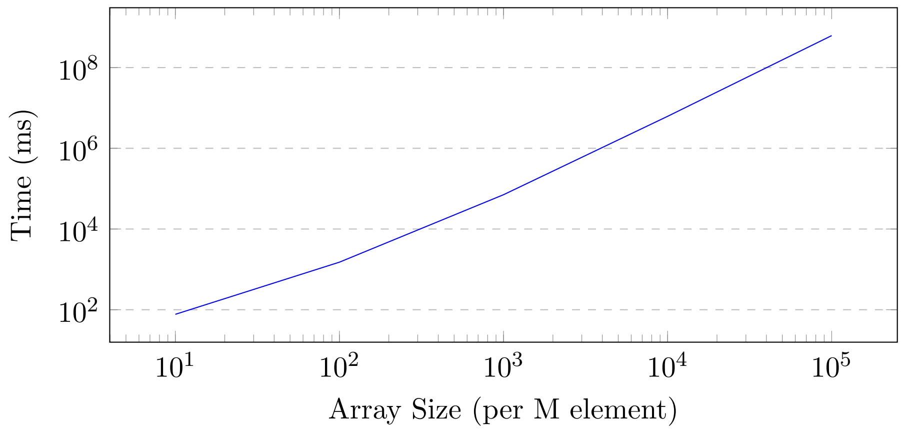

Figure 2: Insertion Sort with different array sizes

Compare to the selection sort, the insertion sort algorithm is more efficient when the array is partially sorted. Additionally, the insertion sort is considered more stable than the selection sort as it does not change the order of the elements that has the right order, while the selection sort does sometimes.

Merge Sort

The merge sort alogrithm divides the array into two halves recursively, then merge the two halves in a sorted order. For a array with n elements, the algorithm will divides the array $\log_2 n$ times, until each subarray has only one element. Then, the algorithm will merge the subarrays, which takes $O(n)$ time. Therefore, the time complexity of the merge sort is $O(n \log n)$.

The implementation of the merge sort alogrithm is a little too long to show here, I uploaded the code to the file in github repository. The result of sorting array with different sizes is shown in Figure 3. We can see that the Time per element(T/n) is proportional to the size of the array in a log scale$(\log_{10} n)$, which means $T \propto n \log n$.

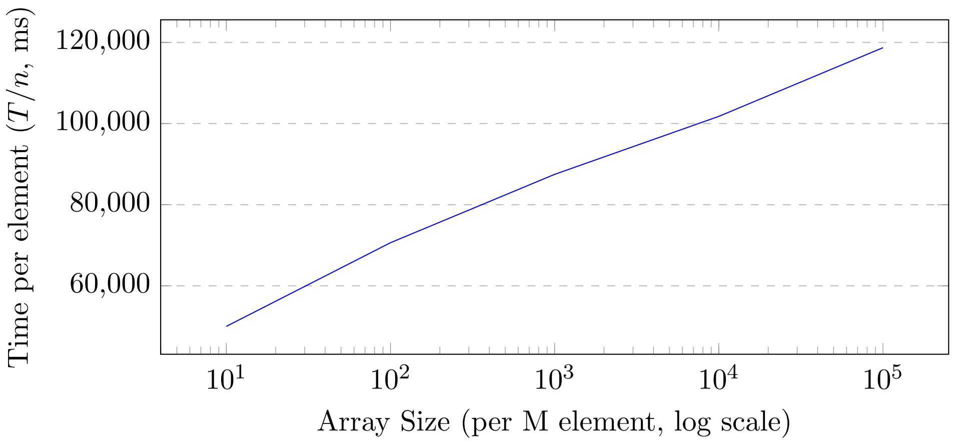

Figure 3: Merge Sort with different array sizes (normalized as T/n)

Quick Sort

The quick sort algorithm start by selecting a pivot element, then partition the array into two parts recursively, one with elements less than the pivot and the other with elements greater than the pivot. The time complexity of the quick sort is the same as the merge sort, which is $O(n \log n)$.

The code snippet shows the implementation of the quick sort algorithm and the result of sorting array with different sizes is shown in Figure 4.

```c
void quick_sort(Array *array, int start, int end) {
  if (start < end) {
    int pivot = partition(array, start, end);
    quick_sort(array, start, pivot - 1);
    quick_sort(array, pivot + 1, end);
  }
}

int partition(Array *array, int start, int end) {
  int pivot = array->array[end];
  int i = start - 1;
  for (int j = start; j < end; j++) {
    if (array->array[j] < pivot) {
      i++;
      swap(&array->array[i], &array->array[j]);
    }
  }
  swap(&array->array[i + 1], &array->array[end]);
  return i + 1;
}
```

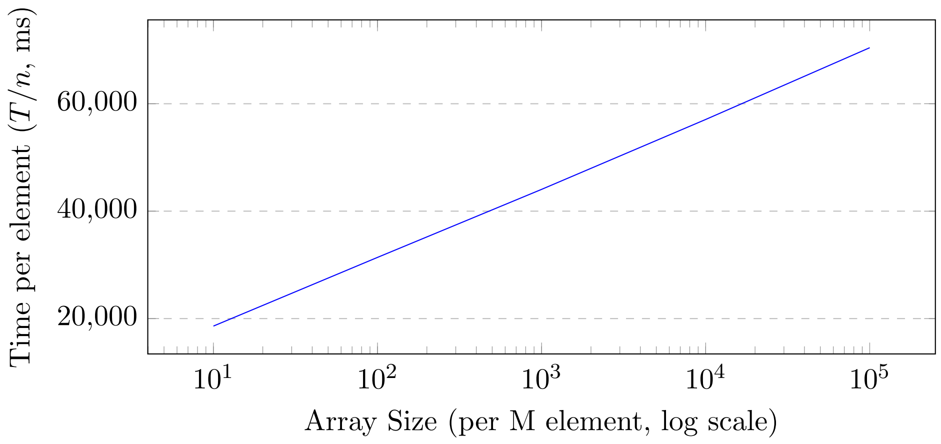

Figure 4: Quick Sort with different array sizes (normalized as T/n)

Optimization of Merge Sort

After implementing the merge sort algorithm, I found that it copies the array reapeatedly when merging the subarrays. By following the suggestion and the resource online, I modified the way of merging the subarrays by using the same array($arrayaux$) to store the sorted elements.

Originally, there are two arrays, right and left, to store the elements of the subarrays, then will be merged into another new array. The new implementation uses the same array to store the sorted elements and merge the subarrays in the same place.

The code snippet is shown at the same file in github repository.