Quicksort Improvements

Josh Wilcox (jw14g24@soton.ac.uk)

March 4, 2025

Contents

1 Quicksort Algorithm

```
QuickSort(arr, start, end) {
    // Check if the start index is less than the end index
    // This ensures that there are at least two elements to sort
    if (start < end){
        // Choose a pivot element from the array
        pivot = ChoosePivot(arr, start, end);

        // Partition the array around the pivot
        // Elements smaller than the pivot will be on the left
        // Elements greater than the pivot will be on the right
        part = Partition(arr, pivot, start, end);

        // Recursively apply the same logic to the left subarray
        // The left subarray consists of elements before the partition index
        QuickSort(arr , start, part-1);

        // Recursively apply the same logic to the right subarray
        // The right subarray consists of elements after the partition index
        QuickSort(arr, part +1, end);
    } else {
        // If the start index is not less than the end index
        // It means the array is already sorted or has only one element
        return;
    }
}
```

2 Standard Quicksort Partitioning Algorithm

```
Partition(arr, pivot, left, right) {
    // Initialize the index of the smaller element
    i = left - 1;
    // Traverse through all elements in the array from left to right-1
    for (int j = left; j <= right-1; j++){
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            // Increment the index of the smaller element
            i++;
            // Swap the current element with the element at index i
            // Sends arr[j] further to the left
            swap(arr[i], arr[j]);
        }
    }
    // Swap the pivot element with the element at index i+1
    // The pivot will be in the correct position
    swap(arr[i+1], arr[right]);
```

```
    // Return the partitioning index
    return i+1;
}
```

3 Hoare Partitioning Scheme

```
Partition(arr, pivot, left, right){
    // Initialize the index of the smaller element
    i = left - 1;
    // Initialize the index of the larger element
    j = right + 1;
    // Loop indefinitely until a return statement is encountered
    while (True){
        // Decrement j until an element less than or equal to the pivot is found
        do {
            j--;
        } while (arr[j] > pivot);

        // Increment i until an element greater than or equal to the pivot is found
        do {
            i++;
        } while (arr[i] < pivot);

        // If the indices have not crossed, swap the elements at i and j
        if (i < j) {
            swap(arr[i], arr[j]);
        } else {
            // If the indices have crossed, return the partition index j
            return j;
        }
    }
}
```

4 Improving Quicksort Partioning

4.1 Choose "Median"

- Find the median of the first, middle, and last element of the array
  - Means it is somewhat likely for the pivot to be close to the median of the whole array

4.2 Random Selection

- Randomly selecting pivots can also be good but not all the time

4.3 Shuffling

4.3.1 Knuth Shuffle

```
KnuthShuffle(arr) {
    // Get the length of the array
    int n = arr.length;
    // Loop through the array from the last element to the first
    for (int i = n - 1; i > 0; i--) {
        // Generate a random index between 0 and i (inclusive)
        int j = (int) (Math.random() * (i + 1));
        // Swap the element at index i with the element at the random index
        swap(arr, i, j);
    }
}
```

5 Multi-Pivoting

- Using multi pivots, number of comparisons remains the same
- Number of swaps is also the same
- Reason for improved performance is due to **fewer cache misses**