Analysis of Algorithms

BLG 335E

Project 1 Report

Mustafa Can Çalışkan
caliskanmu20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 21.11.2023

1. Implementation

1.1. Implementation of Naive QuickSort

1.1.1. Implementation Details

Implementation details of the program are as follows:

- `"main"` function:
  - Reads city population data from a CSV file.
  - Calls "quick_sort" function based on provided arguments.
  - Writes the sorted data to a new CSV file.
  - Measures execution time in nanoseconds and prints it.
- `"quick_sort"` function:
  - The function checks if start_index is less than end_index, ensuring there is more than one element to sort in the current partition of the array. Inside this check, the partition function is called to determine the pivot_index. This partition function separates the array into two parts, elements smaller than the pivot on the left and elements larger than the pivot on the right. After obtaining the pivot_index, the array is recursively sorted: The quick_sort function is called for the left partition (elements less than the pivot), passing the same array, start_index, and pivot_index - 1 as the new end_index. The quick_sort function is called for the right partition (elements greater than the pivot), passing the same array, pivot_index + 1 as the new start_index, and end_index. This process continues recursively, dividing the array into smaller partitions until each partition contains only one element or is empty. As the partitions are sorted individually and merged, the entire array becomes sorted.

```
void quick_sort(vector <pair<string, int>> &arr, int start_index, int
    end_index, bool v) {
    if (end_index > start_index) {
        int pivot_index = partition(arr, start_index, end_index, v);
        quick_sort(arr, start_index, pivot_index - 1, v);
        quick_sort(arr, pivot_index + 1, end_index, pivoting_strategy,
            v);
    }
}
```

- "partition" function:
  - Uses the last element pivot strategy. It initializes an index current_index initially set to (start_index - 1). A loop runs from start_index to end_index to iterate through the elements within the specified range. During each iteration, it checks if the current element (arr[j]) is smaller than the pivot. If an element smaller than the pivot is found, it increments the current_index and swaps the elements at indices current_index and j, effectively placing smaller elements to the left of the pivot. Finally, it places the pivot element in its correct position by swapping the element at index (current_index + 1) with the pivot (arr[end_index]).

```
int partition(vector <pair<string, int>> &arr, int start_index, int
    end_index, bool v) {
    pair<string, int> pivot_pair = arr[end_index];
    int current_index = start_index - 1;
    for (int j = start_index; j <= end_index; j++) {
        if(arr[j].second < pivot_pair.second) {
            current_index++;
            swap_pairs(arr[current_index], arr[j]);
        }
    }
    swap_pairs(arr[current_index + 1], arr[end_index]);
    if (v) {
        write_log_file(arr, pivot_pair.second, current_index + 1);
    }
    return (current_index + 1);
}
```

- Other helper functions:
  - "read_pairs_from_csv" function reads city population data from a CSV file.
  - "write_pairs_to_csv" function writes sorted data to a CSV file.
  - "write_log_file" function logs pivot and array at each step to a log file.
  - "swap_pairs" function swaps two pairs.

1.1.2. Recurrence Relation, Time and Space Complexity

The recurrence relation for regular quicksort is: $T(n) = T(k) + T(n-k-1) + O(n)$ [$T(k)$ and $T(n-k-1)$ represent sorting time of sub arrays, $O(n)$ denotes the partition time.]. If we assume n is a power of 2, recurrence relation becomes $T(n) = 2T(n/2) + O(n)$. The

worst case time complexity is $O(n^2)$. This occurs when the smallest or largest element is selected as the pivot.

The best case time complexity is $O(nlogn)$. This occurs when given array is divided roughly in half at each step.

Since we are not using any extra space in the algorithm, the space complexity is $O(n)$ (Only array size).

1.2. Implementation of QuickSort with Different Pivoting Strategies

1.2.1. Implementation Details

Implementation details after added different pivoting strategies are as follows:

- "partition" function has changed:
  - Random and median pivoting selections have added. First approach is used as last pivoting. After choosing pivot, partition function continues to work as before.

```
switch (pivoting_strategy) {
    case 'l':
        break;
    case 'r': // Random approach
        swap_pairs(arr[(start_index + rand() % (end_index -
            start_index + 1))], arr[end_index]); break;
    case 'm': // Median approach
        swap_pairs(arr[generate_median_of_random_indices(start_index,
            end_index)], arr[end_index]); break;
    default:
        cout << pivoting_strategy << " is an unidentified
            strategy. Program terminated." << endl;
        exit(1);
}
```

- "generate_median_of_random_indices" function is implemented. It finds the median of three random indices.
- Other functions remain unchanged.

1.2.2. Recurrence Relation, Time and Space Complexity

The last element pivot strategy and random pivot strategy share the same recurrence relation as regular quicksort, whereas the median of three pivot strategy, by selecting the median element, has a worst-case time complexity of $O(nlogn)$.

Since the utilized space remains unchanged, the space complexity remains the same as regular quicksort.

	Population1	Population2	Population3	Population4
Last Element	350934397 ns	5044147475 ns	2654739558 ns	10293744 ns
Random Element	11078973 ns	23685171 ns	27453752 ns	32056450 ns
Median of 3	9889773 ns	19147483 ns	10253159 ns	11276338 ns

Table 1.1: Comparison of different pivoting strategies on input data (k = 1 is assumed.).

1.2.3. Discussions

Since `"v"` argument is not used on the experiment, elapsed time has decreased significantly.

In the last element pivot strategy, selecting the largest element as the pivot does not alter the pivot's position, meaning the array won't divide into subarrays. This results in increased complexity, reaching $O(n^2)$. As Population4 differs from the other datasets by being unordered, it operates faster using the last element pivot strategy.

The random pivot strategy might select the highest element in sorted lists (worst case), while the median of 3 strategy is often more inclined to choose the middle element. Consequently, the median of 3 strategy operates faster in sorted lists (Population1 and Population2) compared to the random pivot strategy.

1.3. Hybrid Implementation of Quicksort and Insertion Sort

1.3.1. Implementation Details

Implementation details after hybrid implementation of quick sort and insertion sort are as follows:

- `"quick_sort"` function is changed:
  - Sorts a given array based on pivot strategy until the threshold k is reached.
  - When the size of the sub arrays reaches the threshold, the insertion sort starts.
  - If k is given as 1, naive approach continues. (Based on homework description, I assumed that naive approach contains all pivot strategies.)

```
void quick_sort(vector <pair<string, int>> &arr, int start_index, int
    end_index, char pivoting_strategy, int k, bool v) {
    if (end_index - start_index + 1 > k) {
        int pivot_index = partition(arr, start_index, end_index,
            pivoting_strategy, v);
        quick_sort(arr, start_index, pivot_index - 1,
            pivoting_strategy, k, v);
        quick_sort(arr, pivot_index + 1, end_index, pivoting_strategy,
            k, v);
    } else {
```

```
            insertion_sort(arr, start_index, end_index);
        }
    }
```

- `"insertion_sort"` is added.
  - Function takes in a vector of pairs arr and the indices start_index and end_index to indicate the portion of the array to be sorted. It initializes a loop that iterates from the second element of the specified portion of the array. Each element, starting from the second one, is considered as a key to be placed in its correct sorted position. Within the loop, the value of the current element is stored in key, and an index j is set to the position just before the current element. It checks elements to the left of the current element and moves larger elements one position to the right to make space for the key. Finally, it places the key in its correct position within the sorted sequence and continues this process until all elements in the specified portion of the array are sorted.
- Other functions remain unchanged.

1.3.2. Recurrence Relation, Time and Space Complexity

Threshold (k)	Population4
2	27860385 ns
4	9807765 ns
8	10057449 ns
16	23900304 ns
32	11873187 ns
64	51163474 ns
128	62799759 ns
256	75106800 ns
512	89900966 ns

Table 1.2: Comparison of different thresholds on the hybrid QuickSort algorithm on input data (pivoting_strategy = 'l' is assumed.).

1.3.3. Discussions

Again, since `"v"` argument is not used on the experiment, elapsed time has decreased significantly.

Lower thresholds (2, 4) surprisingly exhibit slower execution times compared to thresholds like 8 and 16, indicating potential inefficiency with small threshold values. Thresholds 8 and 16 showcase notably faster execution times, highlighting the advantage

of employing quicksort for larger segments of the dataset. Thresholds 32 and 64 display fluctuating execution times, with a noticeable increase at 64, indicating diminishing efficiency of Insertion Sort as the threshold grows. Unexpectedly, higher thresholds (128, 256, 512) lead to significantly increased execution times, implying sub optimal sorting strategies at these thresholds. Therefore, when devising solutions, it’s crucial to analyze our dataset and pinpoint a k value that minimizes the algorithm’s runtime, measured in nanoseconds.