Analysis of Algorithms

BLG 335E

Project 1 Report

Ekin Taşyürek
tasyurek19@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 QuickSort with Different Pivoting Strategies

In this project, we are required to write a program that sorts the provided cities based on their population using various sorting algorithms and strategies.

First, we take the arguments from the command line (input file name, pivoting strategy, threshold, output file name, and whether verbose exists). We read the city information and put it into a City vector. We define City as a data structure that holds a city name and population. Then we sort these cities using QuickSort.

We have three different pivoting strategies: Last Element, Random Element, and Median of 3.

- **Last Element**

  The last element pivot strategy is a strategy that always selects the last element of the array as the pivot. This strategy uses the element at the end of the array as the pivot before each division. When the array is nearly sorted, this strategy becomes less favorable.

  Recurrence relation: $T(n) = T(n-1) + O(n)$

  O(n) represents the time taken in the partitioning step.

  Time complexity: $O(n^2)$

  Space complexity: $O(n)$

  Maximum depth of the recursion stack is n, this occurs when the recursion is unbalanced.

- **Random Element**

  If the user determines the random selection strategy, a random element in the array is swapped with the last element before each division. This ensures that a different pivot element is selected each time.

  Recurrence relation: $T(n) = T(q) + T(n-q-1) + O(n)$

  q is the position of the pivot after partitioning. T(q) represents the left subarray, and T(n-q-1) represents the right subarray.

  Time complexity: $O(nlogn)$

  Space complexity: $O(logn)$

  The space complexity is determined by the maximum depth of the recursion stack. In the average case, the depth of the recursion tree is $logn$

- **Median of 3**

  The median selection strategy selects one of three different random elements of the array and uses the median of these three elements as the pivot. This can sometimes improve performance in bad situations, giving us a more balanced split than the random element strategy.

  Recurrence relation: $T(n) = T(q) + T(n - q - 1) + O(n)$

  Time complexity: $O(nlogn)$

  Space complexity: $O(logn)$

	Population1	**Population2**	**Population3**	**Population4**
Last Element	144697000 ns	2134767667 ns	902100459 ns	21341417 ns
Random Element	26829750 ns	23395166 ns	18705875 ns	27872000 ns
Median of 3	26294250 ns	26282250 ns	14450000 ns	26925625 ns

Table 1.1: Comparison of different pivoting strategies on input data.

From the table 1.1, we can see that the running time of random element and median of 3 pivoting strategies are much better compared to last element pivoting strategy.

The last element strategy underperforms on nearly ordered arrays (Population2 is ordered) because it makes a predictable choice by always selecting the last element of the array as the pivot. In contrast, the random element strategy offers more balanced splits and generally better average time complexity by choosing a random element as the pivot each time. The median of 3 strategy, on the other hand, exhibits a more resilient performance against bad situations by choosing the median among three random elements.

1.2. Hybrid Implementation of Quicksort and Insertion Sort

From the command line, we take the threshold. According to the threshold, a sorting algorithm is chosen. If the number of cities is bigger than our threshold, we use the QuickSort algorithm. If not, we use the Insertion Sort algorithm.

On small arrays or nearly sorted arrays the performance of Insertion Sort is better, but on big arrays, QuickSort runs faster.

- **QuickSort**

  This algorithm selects a pivot element, divides the other elements into two sub-arrays according to this pivot, and performs the sorting by repeating this process on the sub-arrays.

  Average-case time complexity: $O(nlogn)$

  Worst-case time complexity: $O(n^2)$

In special cases such as poorly chosen pivots or an already sorted list, Quicksort's performance may decrease.

Recurrence relation: $T(n) = T(n-1) + O(n)$

Space complexity: $O(logn)$

- **Insertion Sort**

  This algorithm offers a simpler approach and performs the sorting process by comparing elements sequentially and swapping them if necessary.

  Average-case time complexity: $O(n^2)$

  Worst-case time complexity: $O(n^2)$

  It can be effective in situations like small data sets or nearly ordered lists.

  Recurrence relation: $T(n) = T(n-1) + O(n)$

  Space complexity: $O(1)$, in space algorithm

Threshold (k)	1	25	100	1000	2000
Population4	29022042 ns	29877375 ns	29029000 ns	30338209 ns	26990500 ns

Threshold (k)	5000	10000	12000	14000	15000
Population4	26532042 ns	28050459 ns	25189750 ns	586657251 ns	565576750 ns

Table 1.2: Comparison of different thresholds on the hybrid QuickSort algorithm on input data.

From the table 1.2, we can see that with thresholds over 13807 (number of inputs), our algorithm runs much slower. So we can say that QuickSort runs much faster than Insertion Sort. Population 4 is completely randomized, it's not ordered/nearly ordered so we can say that our data set is big, since we know QuickSort's performance is better on big data sets.