Analysis of Algorithms

BLG 335E

Project 1 Report

Fatih Baskın
150210710
baskin21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 26.11.2023

1. Implementation

1.1. Implementation of QuickSort with Different Pivoting Strategies

I have implemented the QuickSort as a class. Inside this class, there are two private variables that are important. Those are `mThreshold` and `mStrategy`. They are kept for insertion sort threshold and for pivoting strategy. Those parameters determine how the QuickSort works.

In the first part of the homework, a regular partition function that selects the rightmost index as the pivot was implemented. That function's name is `basicPartition` and its implementation is shown below:

```
int QuickSort::basicPartition(std::vector<City *> &vect,
                              int left_idx, int right_idx) {
    int pivot_idx = left_idx;
    int pivot = vect[right_idx]->mPopulation;
    int curr_idx = left_idx;
    while (curr_idx < right_idx) {
        if (vect[curr_idx]->mPopulation < pivot){
            City *temp = vect[curr_idx];
            vect[curr_idx] = vect[pivot_idx];
            vect[pivot_idx] = temp;
            pivot_idx++;
        }
        curr_idx++;
    }
}
```

Essentially, it compares the populations of the cities pointed by the elements in the array and instead of swapping City objects, it swaps the pointers. Swapping cities is a bad idea since each City object has a name and it is a string, swapping requires string copying which adds overhead.

To implement different strategies, I have written a new partition function called `strategyPartition` which selects the pivot element accordingly with its strategy and then swaps the last element of the given sub-array with the pivot element to recycle the previously described partition code.

```
int QuickSort::strategyPartition(std::vector<City *> &vect,
                                 int left_idx, int right_idx) {
int select_pivot;
switch (mStrategy) {
case PivotStrategy::Median:
    select_pivot = selectMedianPivot(vect, left_idx, right_idx);
    break;
case PivotStrategy::Right:
    select_pivot = right_idx;
    break;
case PivotStrategy::Random:
    select_pivot = selectRandomPivot(left_idx, right_idx);
    break;
}
City *temp = vect[select_pivot];
vect[select_pivot] = vect[right_idx];
vect[right_idx] = temp;
int partition_pivot = basicPartition(vect, left_idx,
                                     right_idx);
// LOG.TXT PART
return partition_pivot;
}
```

Here, `selectRandomPivot` function is just a random number selector between the ranges `left_idx` and `right_idx` (inclusive). But on the other hand, `selectMedianPivot` chooses three random indexes compares them, and chooses their median. Since its code will not fit in the report page, I will talk about some caveats of this function. It directly chooses the right index if the sub-array size is 2. When its sub-array size is 3, it will set the left index to index1, the middle index to index2, and the right index to index 3. If the sub-array size is greater than 3, then random numbers are really chosen. First, index 1, then by being careful that index 2 is not equal to index 1, index 2 is chosen, then finally, with being careful that index 3 is not equal to index 1 and 2, index 3 is chosen randomly. Then the median is determined using extensive if-checks.

In the end, the quick sort function with strategy is written It is called `strategyQuickSort`. It incorporates the `strategyPartition` with strategies which is explained above. It is a recursive function, and the base case is if the sub-array size is 1 or less or in the programming sense, the left index is equal to the right index since an array with cardinality 1 is already sorted. The code looks like this:

```
void QuickSort::strategyQuickSort(std::vector<City *> &vect,
                                  int left_idx, int right_idx)
{
    if (left_idx >= right_idx)
        return;
    int pivot_idx = strategyPartition(vect, left_idx,
                                      right_idx);
    if (left_idx < pivot_idx - 1)
        strategyQuickSort(vect, left_idx, pivot_idx - 1);
    if (pivot_idx + 1 < right_idx)
        strategyQuickSort(vect, pivot_idx + 1, right_idx);
}
```

1.1.1. Recurrence Relation of QuickSort with Different Pivoting Strategies

In each step, the function will do a linear time pivoting and then divide the array into two by this pivot. Then it calls this function for the left of the pivot and the right of the pivot.

In the ideal case, the pivot is in the middle, but it is random for median and random pivoting strategies and dependent on the input for the last element strategy. If we say the pivot index is “p" then the recurrence relation will look like this:

$$T(n) = T(p) + T(n - p - 1) + O(n)$$

1.1.2. Time and Space Complexity of QuickSort with Different Pivoting Strategies

Any quick sort algorithm doesn’t use any auxiliary space, they do their sorting “in-place" meaning they don’t require extra space. Space complexity is $\Theta(1)$. But unfortunately they are not “stable" sorts since they do swapping operations without saving the elements’ relative order. For example, if there is an element that is on the right-hand side of the array, let’s say it pivot, it will jump to the left of all elements that are equal to the pivot.

The worst-case complexity is $O(n^2)$ if the pivot is the first element in each iteration, then the QuickSort is $T(n) = T(0) + T(n-1) + \theta(n) = T(n-1) + \Theta(n) = O(n^2)$.

Best case complexity is when the pivot point is in the middle, then the complexity becomes $O(nlogn)$, calculated from the recurrence relation $T(n) = T(n/2) + T(n/2) + \Theta(n) = 2T(n/2) + \Theta(n) = O(nlogn)$ from the master theorem since $n^{log_2 2} = n^1$.

Since the last element method complexity is dependent on input if the input has some sorted subsections, then its complexity will go near On^2 since the pivot will be the last index, and the recurrence relation will be the worst-case scenario.

Random element and median of 3 strategies choose the pivot element randomly, they still can be very unlucky, and in each iteration, they select the element that will go the last place but it is extremely unlikely $(\frac{1}{n!})$ and most of the cases, it selects a element somewhere in between, which still makes the complexity $O(nlogn)$ but on a different logarithm base since the depth of recurrence tree is depending on how the tree is split. Let's say in each iteration, the array is partitioned into $\frac{2n}{3}$ and $\frac{1n}{3}$, then the complexity will be $O(nlog_{\frac{3}{2}}n)$ since the depth of the recurrence tree is $log_{\frac{3}{2}}n$.

On large datasets, I expect the median of 3 to work better than the random element since the random element can choose a pivot element whose pivot index would not partition the array well, but in the median of 3 cases, it is extremely unlikely since each 3 of those randomly selected indexes should be on edges for this to occur. Mathematically, it is comparing $\frac{1}{n} > \frac{3!}{n(n-1)(n-2)}$.

	Population1	**Population2**	**Population3**	**Population4**
Last Element	123636293 ns	1571281946 ns	985585977 ns	4495911 ns
Random Element	3898045 ns	4253853 ns	3776777 ns	4315319 ns
Median of 3	3331343 ns	5980235 ns	3497275 ns	5202745 ns

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

As expected, the last element pivoting strategy works slower than the other strategies. If the input is sorted or semi-sorted, it is highly likely that a pivot will be on the leftmost or rightmost point and the partition will fail to split the array in a balanced manner.

In some cases, the random element did better than the median of 3. This probably occurred due to randomizing working well for these inputs for the random element and the median of 3 worked slower due to it having some overhead of randomization and pivot selection while choosing a pivot.

1.2. Hybrid Implementation of Quicksort and Insertion Sort

In this part, I have written the insertion sort algorithm first. It works in $O(n^2)$ time but if the input has some semi-sorted parts, it starts to get closer to linear time. It works by assuming the sub-array on the left-hand side is already sorted and the current element is being put in place by shifting it to the left. It works since an array with only one element is already sorted and in each iteration will expand the array by one by placing an element in its correct place. The code is:

```
void QuickSort::insertionSort(std::vector<City *> &vect,
                              int left_idx, int right_idx) {
    for (int i = left_idx + 1; i <= right_idx; i++) {
        int key = vect[i]->mPopulation;
        int key_idx = i - 1;
        while (key_idx >= left_idx &&
        vect[key_idx]->mPopulation > key) {
            City *temp = vect[key_idx + 1];
            vect[key_idx + 1] = vect[key_idx];
            vect[key_idx] = temp;
            key_idx -= 1;
        }
    }
}
```

After the implementation of `insertionSort`, I have modified the `strategyQuickSort`. The new function is called `quickSort`. If the size of the subarray is smaller than the threshold of insertion sort, then the insertion sort is called then the function returns and quick sort will not be called. But if the subarray size is bigger than the threshold, then partition and quicksorts will be called. The code is below:

```
void QuickSort::quickSort(std::vector<City *> &vect,
                          int left_idx, int right_idx) {.
    if (left_idx >= right_idx)
        return;
    if (right_idx - left_idx + 1 <= mThreshold) {
        insertionSort(vect, left_idx, right_idx);
        return;
    }
    int pivot_idx = strategyPartition(vect, left_idx,
                                      right_idx);
    if (left_idx < pivot_idx - 1)
        quickSort(vect, left_idx, pivot_idx - 1);
    if (pivot_idx + 1 < right_idx)
        quickSort(vect, pivot_idx + 1, right_idx);
}
```

1.2.1. Recurrence Relation of the Hybrid QuickSort Algorithm

There are two recurrence relations, depending on the input size “n" and insertion sort threshold “k". Pivot is denoted as “p".

$$T(n) = \begin{cases} T(p) + T(n-p-1) + \Theta(n), & \text{if } n > k \\ O(n^2), & \text{otherwise} \end{cases}$$

1.2.2. Time and Space Complexity of the Hybrid QuickSort Algorithm

The space complexity of this algorithm is still the same, both quick sort and insertion sort algorithms are “in-place" meaning they sort without a need for auxiliary space, and their space complexity is $\Theta(1)$. Insertion sort is a “stable" sorting algorithm but since the quick sort is not, the overall algorithm is not stable.

When the sub-array size is less than or equal to threshold k, insertion sort will take place and its worst-case complexity is $O(n)$. If the array is sorted in the reverse order, it will do $i-1$ iterations to put the next element in its place in each iteration and there are $n-1$ iterations. So, the number of iterations:

$$\Sigma_{=1}^{n-1} i - 1 = 1 + 2 + 3 + \cdots + n - 1 = \tfrac{n(n-1)}{2} = O(n)$$

But insertion sort shines in the cases when the input is semi-sorted in the right order, since there are a few items to put its place, the inner iteration will work only a few times so its complexity would be $O(n)$. In cases like that, insertion sort makes more sense than quick sort but the probability of semi-sorted input is very low for large thresholds. But in the small thresholds, the probability of semi-sorted input is relatively high, and loss from input not being semi-sorted is not very high, so for small thresholds, the quick sort algorithm with the hybrid approach should run faster.

Experiments were done with the last index pivoting strategy to compare the threshold values fairly.

Threshold (k)	1	2	3	4	5	10
Population4	4444483	4150155	4211341	4476736	4768211	5712408
Threshold (k)	20	50	100	200	500	1000
Population4	5352666	6028666	8717930	13630244	24051756	50607721

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

For comparison, the last element method with no insertion sort (k = 1) lasted for “4444483 ns". The function’s work time starts to diverge from the non-hybrid approach is about 10 or 20 elements. As discussed above, if the input is semi-sorted, insertion sort might work faster than regular quick sort, but in the case of randomized inputs, insertion sort’s overhead is not that much for small input sizes.

The outcome of this experiment coincides with the hypothesis described above. For the input data which is semi-sorted, hybrid approach can work better but if the input is in the reverse order then insertion sort will work slower.

For the real-life data, we can’t make sure how the input is coming, it can be sorted, in can be random or it can be semi-sorted. But human intuition will enter data in a sorted manner for small input sizes and if this input schema cumulative, lots of batches of small sorted inputs will be provided and the hybrid approach might work very well in the cases like this.