CS-201 2023-2024 Spring
Section:2 Homework: #2

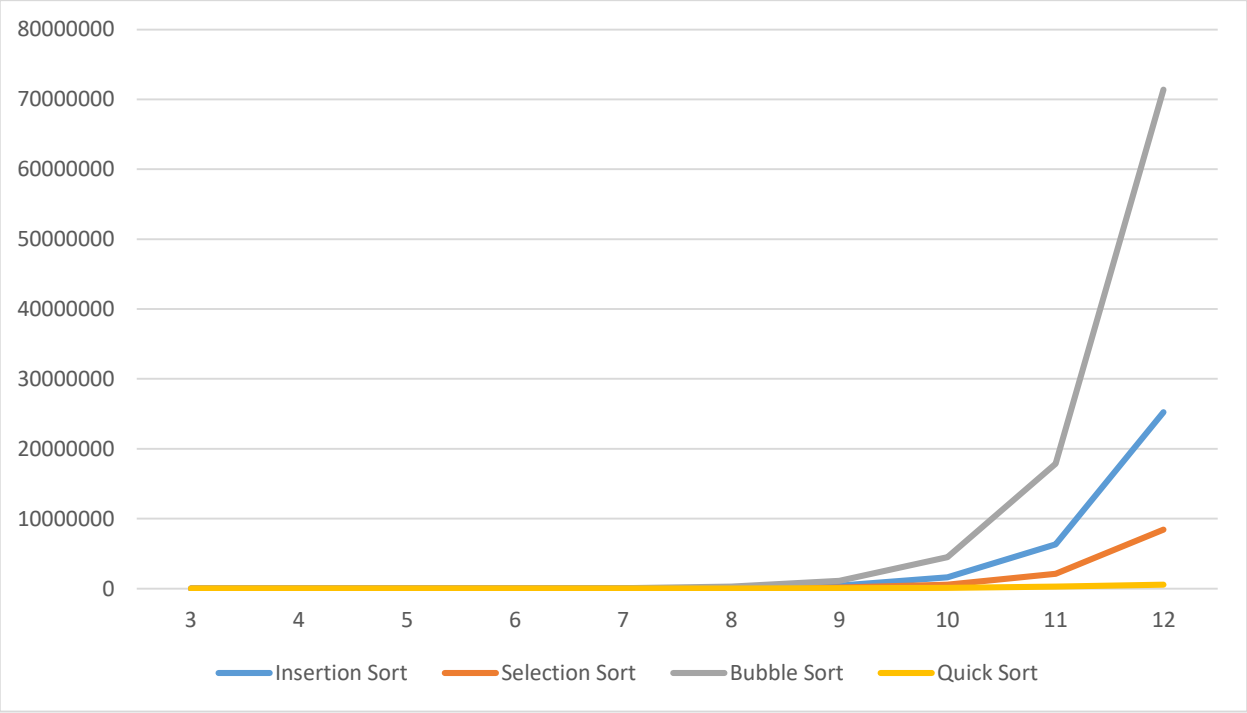
Halil Arda Özongun
22202709
08/04/2024

Part1: Which Algorithm?

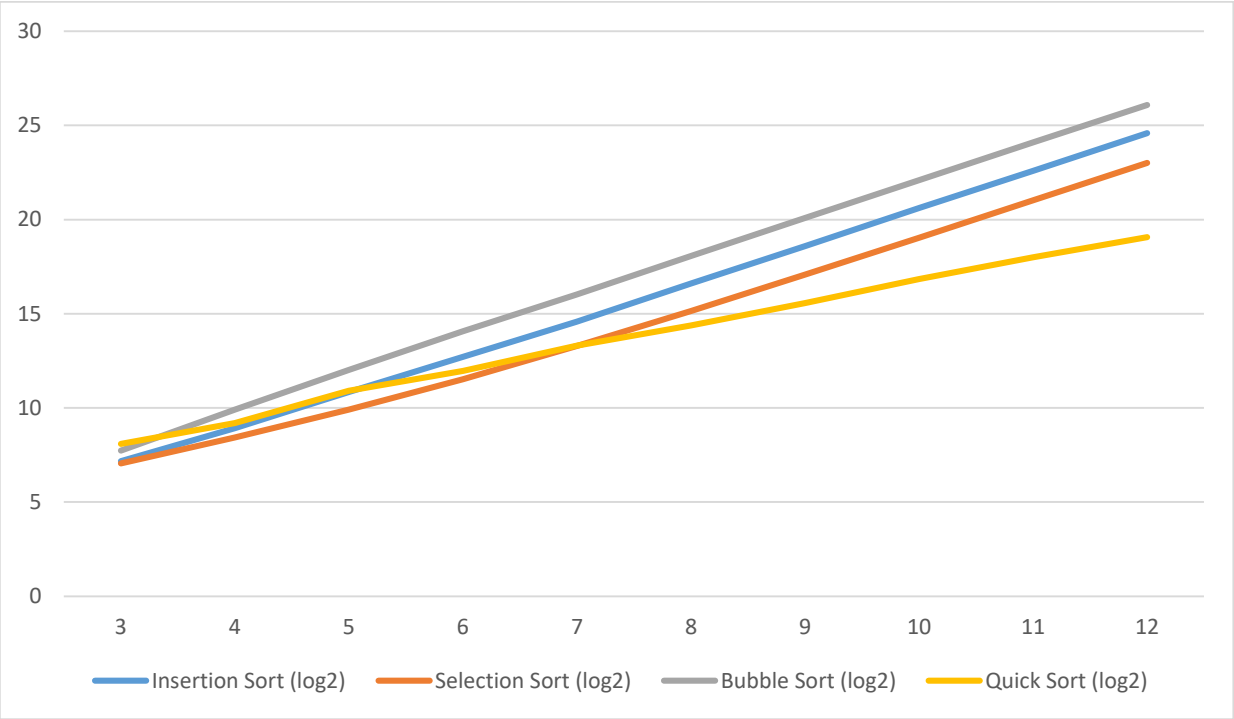
Table1: Time estimations as seconds for Randomly Generated Arrays

n	Insertion Sort	Selection Sort	Bubble Sort	Quick Sort
2^3	145	133	214	273
2^4	489	345	965	585
2^5	1836	961	4138	1929
2^6	6726	2961	17244	4021
2^7	24652	10033	66513	10169
2^8	99476	36465	274868	21386
2^9	397170	138481	1110529	48801
2^{10}	1595131	539121	4484692	117186
2^{11}	6323148	2126833	17852020	259693
2^{12}	25252136	8447985	71412635	552253

Plot 1: Time Estimation Comparison of Sorting Algorithms for Randomly Generated Arrays



Plot 2: Logarithmic (Base-2) Time Estimation Comparison of Sorting Algorithms for Randomly Generated Arrays



Discussion of results:

Let's begin with theoretical background. selection sort always has a time complexity of O(n^2) for best, average, and worst cases. Insertion sort, exhibits a best-case time complexity of O(n), which occurs when the data is already sorted or nearly sorted. However, in average and worst case complexities are O(n^2), which occurs when the data is not sorted or is far from being sorted. Bubble sort, generally has a O(n^2) time complexity, which is in its average and worst cases. It has a best case with O(n) which occurs when array is already sorted. Quick sort, on the other hand, has a time complexity of O(nlogn) on average, while the worst case with O(n^2) which occurs when the first element is the value we choose as the pivots. In this task, arrays are randomly generated so we are interested in the average case.

When we look at the results, we see that it is difficult to distinguish between algorithms for small data sets. This is because for small numbers O(nlogn) and O(n^2) are close to each other. As the size of the arrays increases, quick sort performs better than other algorithms in terms of time efficiency. The fact that the time complexity of quick sort is O(nlogn) in average cases has enabled it to sort larger datasets more efficiently.

For insertion sort, selection sort, bubble sort, all theoretical results for average case are O(n^2). The empirical results agree with the theoretical results. Indeed, as can be seen from both the table and the graph, they start with a different initial value and then increase linearly by the square of n. Although the growth rate of all of them is O(n^2), it is quite normal for their values to be different. It is important for each algorithm to be proportional to n^2. Which coefficient they are proportional to depends on the number of swaps and the number of comparison they make. Bubble sort, which does a lot of swaps, therefore takes even more time than the others since swap cost is 15 times bigger than comparison cost.

Conclusion:

When Bilkent University workers sorts the trees, it will not make much difference which sorting algorithm is used for a small number of trees. However, as the number of trees increases, University should use the quick sort algorithm in order to reduce both the time and the money paid to the worker.

Part2: An Assumption!

Before the empirical results:

Considering that more than 6% of the arrays are sorted, we can quickly find a solution by putting a very small number of trees in the right place. The sorting method that I believe would be the best to use for this is insertion sort. Insertion sort requires fewer comparisons and fewer swaps in almost sorted arrays. We can think of it as almost like best case. In the best case, insertion sort is O(n), because the inner loop is not executed in most cases.

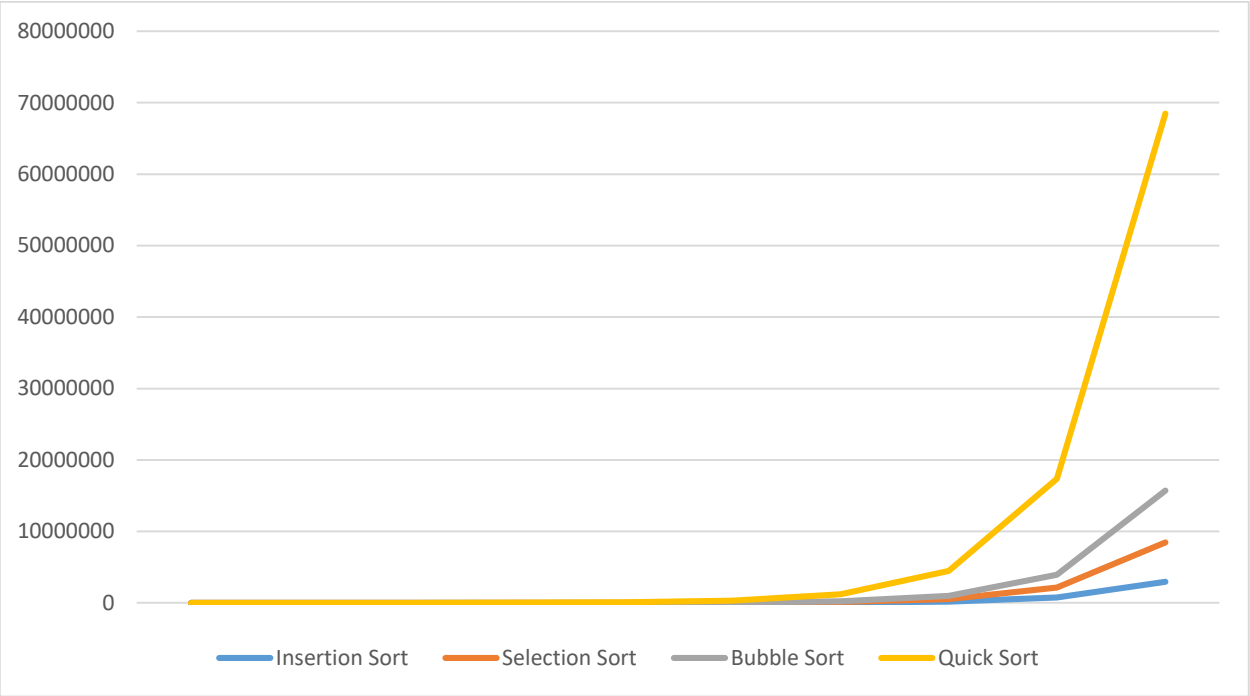
For other sorting algorithms, the selection sort should not change much, because theoretically it is O(n^2) for best, worst, and worst cases. The performance of bubble sort should improve compared to random generated array, but it should still not provide very good performance since the swap cost is high.

The one I expect to perform the worst is quick sort. This is because quick sort is theoretically O(nlogn) for best and average case, but O(n^2) for worst case. In this case it should be almost worst case because the array is almost ordered. This should mean that if the chosen pivot is chosen at the beginning or last, the for loop will in most cases run for nothing, since the pivot tends to be already in its place, and very few data will be in the wrong position.

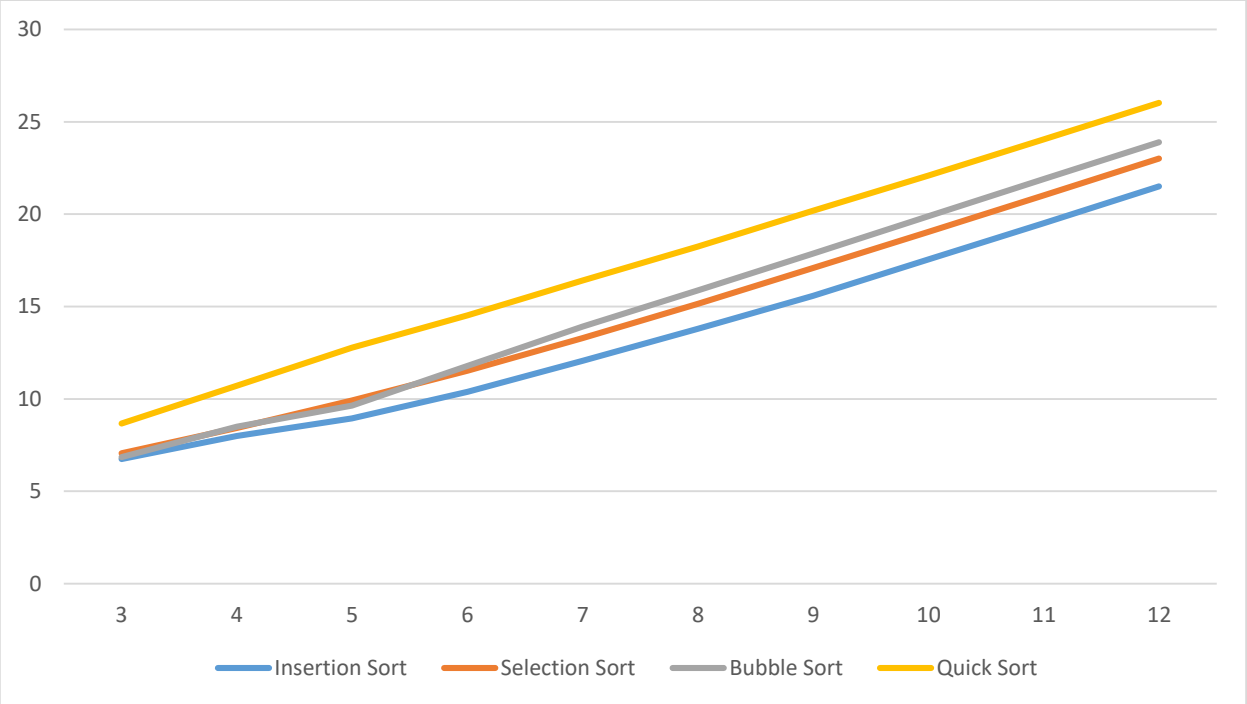
Table 2: Time estimations as seconds for Almost Sorted Arrays

n	Insertion Sort	Selection Sort	Bubble Sort	Quick Sort
2^3	107	133	115	405
2^4	254	345	360	1671
2^5	491	961	793	6964
2^6	1333	2961	3514	23468
2^7	4272	10033	15350	87822
2^8	14185	36465	60509	312590
2^9	48978	138481	238558	1185561
2^{10}	192475	539121	977118	4459979
2^{11}	750830	2126833	3920367	17359514
2^{12}	2964276	8447985	15694001	68456768

Plot 3: Time Estimation Comparison of Sorting Algorithms for Almost Sorted Arrays



Plot 4: Logarithmic (Base-2) Time Estimation Comparison of Sorting Algorithms for Almost Sorted Arrays



Discussion of Results:

The results were generally consistent with the theoretical results. For small data sets they all perform more or less the same, but for large arrays the difference between them is clearly visible.

Insertion sort indeed outperformed all other sorting algorithms, which is consistent with the theoretical results. For selection sort, the best, average and worst cases are all O(n^2) and the results are exactly the same, confirming this. The best case for bubble sort is actually O(n), but it falls behind both insertion and selection sort because it does not reflect the best case for almost sorted bubble sort and also because the cost of swap is much higher than the cost of compare. Nevertheless, it is much less costly than bubble sort applied to a completely randomly generated array.

In my algorithm for quick sort, the last element is always chosen as the pivot. In an almost sorted array, since most of the elements are already in right place, most of the time it will run for nothing and affect the performance badly. Therefore, quick sort performs worse in almost sorted array than in random generated array. In fact, quick sort is even worse than bubble sort among other sorting algorithms.

Conclusion:

If the university administration receives such trees from a place that sends them in almost sorted order, they should use insertion sort. Having an idea of whether the trees are sorted or not has a significant impact on the algorithm we should use.

Part3: Increasing the number of workers.

My Suggested Algorithm:

Increasing the number of workers gave me a few ideas. The first one is to increase the number of pivots in our sorting algorithms. To explain a bit more, doing the same thing we do in each for loop, but for both the highest case and the lowest case, could allow us to split the work into two workers. This approach may not be useful for big O notation. For example, something that can be done in $\log_2 n$ can be done in $\log_3 n$, and since they only give the same result when multiplied by a constant, they actually have the same big o complexity. However, even if the total work time is not reduced, the time of each worker will be reduced, which will bring us closer to the desired result. I have modified two algorithms for this approach, selection sort and quick sort. Perhaps this could also be done for insertion and bubble sort, but as I will write in the conclusion, trying to optimise them may not be that useful for us.

If we need to examine this approach for selection sort a little more, the main purpose of the algorithm is to halve the number of iterations by finding both maximum and minimum in a single iteration, instead of finding only minimum or only maximum. To do this, we find the minimum and maximum elements continuously in a loop and place the maximum element at the rightmost part of the array (where the larger elements are located, smaller than the larger ones) and the minimum element at the leftmost part of the array (where the smaller elements are located, larger than the smaller ones). It is explained a little more with the code below.

For quick sort, the algorithm I am considering aims to find two pivots and divide the array into 3 sub-arrays instead of 2. Instead of leaving the workers alone with the sub-arrays found, the workers aim to progress by helping in all these 3 sub-arrays and dividing the cost. A different approach would be to split the array completely in two and ask each worker to sort each piece in a quick sort, but I would not prefer this.

Choosing two pivots is actually a very intuitive way, it really gives you an idea of what the workers are doing. It gives the workers a somewhat regular path. But another approach could be multithreading. The concept of multithreading seems to me to be very similar to having more than one worker do the same job at the same time. The idea is to use multiple processing units (cores) to speed up the overall sorting process by sorting different parts of the sequence simultaneously. Thinking of each core as a worker will make the analogy a little clearer. In order to do this as code, we need to know the thread library. I won't go further into this topic as it may be out of context.

Code of algorithms: (also exists in main.cpp):

Selection Sort:

```
void selectionSortWithTwoIndex(int*& arr, const int size, int&
costOfFirstWorker, int& costOfSecondWorker) {
    int iteratorRight = 0, iteratorLeft = size - 1;

    while (iteratorRight < iteratorLeft) {
        int minIndex = iteratorRight;
        int maxIndex = iteratorLeft;

        for (int j = iteratorRight; j <= iteratorLeft; j++) {
            costOfFirstWorker += COMPARE_COST;
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
            costOfSecondWorker += COMPARE_COST;
            if (arr[j] > arr[maxIndex]) {
                maxIndex = j;
            }
        }
        costOfFirstWorker = swap(arr[iteratorRight], arr[minIndex]);

        // There is an edge case, if the maxIndex is equal to the
iteratorRight, we need to update the maxIndex
        if (maxIndex == iteratorRight) {
            maxIndex = minIndex;
        }

        costOfSecondWorker = swap(arr[iteratorLeft], arr[maxIndex]);
        iteratorRight++;
        iteratorLeft--;
    }
}
```

I would like to explain the logic in the code a bit:

Initially, two iterators are selected at the rightmost and leftmost part of the array: iteratorRight and iteratorLeft. These iterators are controlled by a while loop, as long as the right iterator is to the left of the left iterator, the process is continued in a while loop. The aim is to find the min and max number in the range searched in the while and change the minimum number to the right iterator and the maximum number to the left iterator. That is, everything before the right iterator is the minimum elements of the list, and everything to the left of the left iterator is the maximum elements of the list. Every value between the two iterators must lie between the min part and the max part (because if they were larger or smaller we would have already selected them and moved them to the required place), and this middle part must be unsorted. Each time we find a max and a min, we have to replace the right iterator and the left iterator respectively and move the iterators one by one towards the center, so that the unsorted part in the middle is not mixed with the sorted parts.

Quick Sort:

```
void partitionForDualWorkerQuickSort(int* arr, int low, int high, int& lp,
int& rp, int& costLeft, int& costRight) {
    // defined dividers, which are used to divide the array into 3 parts,
smaller, between, bigger
    int firstDividerIndex = low + 1;
    int secondDividerIndex = high - 1;
    bool checkIsDone; // a boolean, which checks if compare cost added to any
of the workers

    // We are trying to divide 3, smaller, between, bigger so we must be sure
right pivot is bigger than left pivot
    if (arr[low] > arr[high]) {
        if(costLeft < costRight){
            costLeft += swap(arr[low], arr[high]);
        }else {
            costRight += swap(arr[low], arr[high]);
        }
    }

    // The aim of pivots, are same with the classic quick sort
    int rightPivot = arr[high];
    int leftPivot = arr[low];

    // iterating from the begining untill the second divider
    // second divider shows the end of the middle part, which means beyond the
second driver, all elements are bigger than right pivot so they are already
handled
    for (int iterator = low + 1; iterator <= secondDividerIndex; iterator++) {
        checkIsDone = false;

        // if the element is smaller than left pivot, swap it with the first
divider and increase the first divider
        if (arr[iterator] < leftPivot) {
            costLeft += swap(arr[iterator], arr[firstDividerIndex]); // Cost
for swapping less than pivot attributed to the left worker
            firstDividerIndex++;
            costRight += COMPARE_COST;
            checkIsDone = true;
```

```
        // if the element is bigger than right pivot, swap it with the second
divider and decrease the second divider
        } else if (arr[iterator] >= rightPivot) {

            // trying to find the first element from the end, which is smaller
than right pivot
            while (arr[secondDividerIndex] > rightPivot && iterator <
secondDividerIndex) {
                secondDividerIndex--;
                costRight += COMPARE_COST;
            }
            // after finding the element, swap it with the second divider and
decrease the second dividers index
            costRight += swap(arr[iterator], arr[secondDividerIndex]);
            secondDividerIndex--;
            // if the new iterator is smaller than left pivot, swap it with
the first divider and increase the first divider
            if (arr[iterator] < leftPivot) {
                costLeft += swap(arr[iterator], arr[firstDividerIndex]);
                firstDividerIndex++;
            }
            costLeft += COMPARE_COST;
            checkIsDone = true;

        }

        // if the element is in the middle, compare costs should be added to
both workers
        if(!checkIsDone){
                costRight += COMPARE_COST;
                costLeft += COMPARE_COST;
        }

    }
    // change positions of pivots
    costLeft += swap(arr[low], arr[firstDividerIndex-1]);
    costRight += swap(arr[high], arr[secondDividerIndex+1]);

    // update lp and rp
    lp = firstDividerIndex-1;
    rp = secondDividerIndex+1;
}

void DualWorkerQuickSort(int* arr, int low, int high, int& costOfFirstWorker,
int& costOfSecondWorker) {
    if (low < high) {
        int lp, rp; // Left and right pivots
        partitionForDualWorkerQuickSort(arr, low, high, lp, rp,
costOfFirstWorker, costOfSecondWorker);

        DualWorkerQuickSort(arr, low, lp - 1, costOfFirstWorker,
costOfSecondWorker); // Sorting first part
        DualWorkerQuickSort(arr, lp + 1, rp - 1, costOfFirstWorker,
costOfSecondWorker); // Sorting middle part
        DualWorkerQuickSort(arr, rp + 1, high, costOfFirstWorker,
costOfSecondWorker); // Sorting last part
```

```
        }
}
```

Since the description of the code is already available as a comment in code, I will not explain it too deeply, instead I will talk about the purpose of the code in general. The main purpose of the algorithm is to divide the array into three parts with two pivots, lp and rp, and to sort the three parts of the array into three parts. In order to locate the pivots correctly, a helper method partitionForDualWorkerQuickSort is used. In this method, if an element is below the left pivot, it is moved to the appropriate place by the left worker and the cost is written to the left worker; if the element is above the right pivot, it is moved to the appropriate place by the right worker and the cost is written to the right worker.

Results:

Table 3: Time estimations with new algorithms and their old versions for Randomly Generated Arrays

Array Size (n)	**Selection Sort Cost**	**Selection Sort with Two Index Cost (Total)**	**First Worker Cost (Selection Sort with Two Index)**	**Second Worker Cost (Selection Sort with Two Index)**	**Quick Sort Cost**	**First Worker Cost (Dual Pivot Quick Sort)**	**Second Worker Cost (Dual Pivot Quick Sort)**	**Dual Pivot Quick Sort Cost (Total)**
2^3	133	160	80	80	275	96	91	188
2^4	345	384	192	192	682	243	199	442
2^5	961	1024	512	512	1766	645	544	1190
2^6	2961	3072	1536	1536	4240	1305	1237	2542
2^7	10033	10240	5120	5120	10414	3401	2813	6214
2^8	36465	36864	18432	18432	21924	8150	6133	14283
2^9	138481	139264	69632	69632	50829	17431	14396	31828
2^{10}	539121	540672	270336	270336	109155	36470	31973	68443
2^{11}	2126833	2129920	1064960	1064960	259946	90318	67964	158282
2^{12}	8447985	8454144	4227072	4227072	552123	188909	145785	334694

From the results, splitting into two workers almost does not change the total cost for the selection sort. However, since the number of workers is doubled, the time is halved and the desired result is achieved. I have not converted the times here into costs because it is already clearly seen in the table that the total duration does not change for the selection sort, that is, the price paid does not change, but the duration is halved. And for quick sort, both the time and the total fee to be paid are significantly reduced with this approach.

For quick sort, both the work required by each worker is decreased and the work done by the workers together is decreased. With this new method, hiring more than one worker is more advantageous than hiring only one worker. Looking at these results, dual quick sort is actually the most efficient one as it was thought at first. However, if we compare the workers, we can see that

there is an injustice in the distribution of work among the workers. With a better code or algorithm, this could be overcome and the time of the workers could be kept at a more equal value which means total time required might decrease. In my approach, it is hard to make a worker's workload is more compared to other since they are splitting work in each subarray. So the maximum time required might not be as big as their total time. But if we would use dividing into 2 subarray approach, it might end up with a result that, one worker does almost all job while other does nothing.

Both of the above two approaches are based on the idea of two workers handling the whole data set together. However, we could also divide the set into two parts first and then distribute the work to the workers. For example, for quick sort, we can first split the array directly into two parts, and then have the two split parts sorted by one worker each. Finally, we would have to merge these two sorted parts back together, and for this part we could find an optimised merge operation involving two workers again. In fact, any function could have been used for the sorting function in this method of first splitting in half, sorting the parts and then merging them, and the result here could affect the overall result positively or negatively depending on the time lost in the merge operation.

In addition to the two new sorting I did, I also thought of implementing merge sort with two pivots, because splitting the array directly into two and doing it this way is similar to the approach of classical merge sort in the first place. However, since in merge sort we have to create a new array each time and copy the elements there, the number of swaps and therefore the cost would be too high, so I didn't find this approach logical.

In fact, we could have used the two pivot logic with one worker, and it would have increased the efficiency significantly again. I chose to use these algorithms here in particular because the more pivots here, the more intuitively similar it is to selecting two workers.

Conclusion:

Among these two sorting algorithms, quick sort is already the faster one. With this modification, it is now even faster and more suitable for two workers. It would be nonsense if Bilkent University chooses selection sort among the algorithms, because for high number of trees, selection sort with two workers will still be slower than quick sort with one worker. Therefore, Bilkent University should choose either quick sort or the multithreading approach.