Analysis of Algorithms

BLG 335E

Project 1 Report

İbrahim Karateke
150210705
karateke21@itu.edu.tr

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

1. Implementation

1.1. Naive Implementation of Quicksort

In the beginning of this homework, we are asked to implement a quicksort algorithm without any optimization. Therefore, the quicksort algorithm is going to take a collection of numbers (i.e unsorted vector), then it will manipulate those in order to get an ordered collection of numbers in ascending order. This part starts with the implementation of **part1_QuickSort** function. The code is provided below.

```
void part1_QuickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivotIndex = part1_Partition(arr, low, high);

        part1_QuickSort(arr, low, pivotIndex - 1);
        part1_QuickSort(arr, pivotIndex + 1, high);
    }
}
```

In order to enhance the explanation code of **part1_Partition** function the is also provided below.

```
int part1_Partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}
```

Those provided code snippets embody the Quicksort algorithm, a sorting technique based on partitioning elements within a vector. The **part1_Partition** function selects the last element (arr[high]) as the pivot, then rearranges the sub-collection such that elements smaller than the pivot occupy the left side, and those greater, the right. Utilizing two indices (i and j), it iterates through the sub-collection and swaps elements to appropriately position them relative to the pivot. Once the traversal concludes, it places the pivot in its correct sorted position, returning the index where it's located. On the other hand, the **part1_QuickSort** function recursively divides the vector into smaller sub-collections

using the pivot obtained from **part1_Partition**. It performs partitioning and sorting on these sub-collections until the entire vector is sorted. This recursive process continually refines the partitioning and sorting, ultimately arranging the vector elements in ascending order.

The swap function simply swaps the value of two elements in certain indices. The code of **swap** function is provided below.

```
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

In order to see the initial and final vectors, **printArray** function is also provided. It simply traverses the vector from the first element to the last element of the vector and prints them in the desired format. Related code snippet is also provided below.

```
void printArray(const std::vector<int>& arr) {
    std::cout<<"[ ";
    for (int i : arr) {
        std::cout << i << " ";
    }
    std::cout<<"]";
    std::cout << std::endl;
}
```

During the compilation of this code our **main** function was as following.

```
int main() {
std::cout<<"Initial Array: "<<std::endl;
std::vector<int> arr = {5,2,1,9,3,10,6,8,7,0,4};
printArray(arr);
std::cout<<"Sorted Array: "<<std::endl;

const auto start{std::chrono::steady_clock::now()};
part1_QuickSort(arr,0,arr.size()-1);
const auto end{std::chrono::steady_clock::now()};

printArray(arr);

const std::chrono::duration<double> elapsed_seconds{end - start};
std::cout << elapsed_seconds.count()*1000000000 << " ns"<< std::endl;
return 0;
}
```

Hence, the output after running this code looks like as following.

```
Initial Array:
[ 5 2 1 9 3 10 6 8 7 0 4 ]
Sorted Array:
[ 0 1 2 3 4 5 6 7 8 9 10 ]
900 ns
```

1.2. Implementation of Quicksort Using Different Pivoting Strategies

In this part, we are going to improve our quicksort function by providing different options in terms of pivoting strategies. Since our data is given in .csv format we need to read this data and process it. Furthermore, we have two data which are population and city name and we have to match both two attributes for each entity in the data set. In order to achieve this, we can define a new data type such as **pair**. Then, we are going to keep this data in this pair. This operation is held by **csvToVector** function. It takes the name of the .csv file and our vector where the data will be kept. This function uses the **split** function in order to separate the columns in .csv file and move it to the vector with respect to the delimiter. In this case, the provided delimiter in .csv file is the semicolon (;). Hence, whenever the program encounters with a semicolon in each line, it differentiates the city name and the population and pushes this pair into our vector. Code snippets of those functions are given below.

```
1  std::vector<std::string> split(const std::string &s, char delimiter) {
2      std::vector<std::string> tokens;
3      std::stringstream ss(s);
4      std::string token;
5      while (std::getline(ss, token, delimiter)) {
6          tokens.push_back(token);
7      }
8      return tokens;
9  }
```

```cpp
typedef std::pair<std::string, long long> CityPair;

void csvToVector(std::string filename,std::vector<CityPair> &data){
    std::ifstream file(filename); // CSV file name

    if (!file.is_open()) {
        std::cerr << "Error opening file!" << std::endl;
        return;
    }

    std::string line;
    while (std::getline(file, line)) {
        std::vector<std::string> row = split(line, ';'); // CSV data is
        ↪ semicolon-separated
        if (row.size() == 2) {
            std::string city = row[0];
            long long population = std::stoll(row[1]);
            CityPair cityData = std::make_pair(city, population);
            data.push_back(cityData);
        }
    }

    file.close();
}
```

Quicksort function is very similar to the previous part. The only difference is the fact that we have to determine the pivoting strategy. In addition to the previous quicksort function, this function takes a new parameter, which is **selection**, and input vector type is also changed by the data type that we have defined previously, which is **CityPair**. In this part, we named our quicksort function as **part2_QuickSort**. The code of this function is provided below.

```cpp
void part2_QuickSort(std::vector<CityPair> &data, int low, int high,char selection) {
    if (low < high) {
        int pivotIndex = part2_Partition(data, low, high,selection);

        part2_QuickSort(data, low, pivotIndex - 1,selection);
        part2_QuickSort(data, pivotIndex + 1, high,selection);
    }
}
```

In order to achieve different pivoting strategies, some changes have been made in the partition function from now on this function is going to be called **part2_Partition**. According to this selection parameter, the partition is going to determine the pivoting strategy. There are three different conditions that we have to consider. The new partition function behaves in three different ways. We will go over each case about the fact that what is done on each strategy.

1. **If selection is 'l':**

The parameter 'l' stands for the last index. No different operation is done compare to naive quicksort. It always selects the last index of the collection or the sub-collection.

2. **If selection is 'r':**

The parameter 'r' stands for random element. In this case, any element of our collection is a candidate to be a pivot. It generates a random number between the highest and the lowest indices. Hence, our pivot's index is selected randomly. At the end, the value is in this index and the value in the last index is swapped. Therefore, comparisons between our value in the last index, in fact after swapping it is our random pivot, and the other value begins.

3. **If selection is 'm':**

The parameter 'm' stands for the median of three elements. In this case, it is very likely to select the numbers where the fact that the highest and the lowest values are avoided to become a pivot. Because in this approach, we select three random numbers instead of one random number. Those selected numbers will be in the range of our highest and lowest indices. Once we determine those three numbers, we will reach the values in those indices. Afterwards, thanks to the comparison those three numbers we will determine the median of those three values. At the end, the median of those three is going to be our pivot. Since we know its index, the value is in this index and the value in the last index is swapped. Therefore, comparisons between our value in the last index, in fact after swapping it is our random pivot of the median of three, and the other value begins. Notice that since it is a recursive function in the whole, there are some possibilities where the number of elements in a sub-collection is less than three. In this case, it selects the value in which its index number is the least.

The code snippet of this function is provided below.

```
int part2_Partition(std::vector<CityPair> &data, int low, int high, char selection) {
long long pivot;
    if (selection=='l')
    {
        pivot = data[high].second;
    }

    else if(selection=='r')
    {
        int randomNumber = std::rand() % (high - low + 1) + low;
        int pivotIndex= randomNumber;
        swap(data[pivotIndex], data[high]);
        pivot = data[high].second;
    }
    else if (selection=='m')
    {
        if (high - low + 1 < 3) {
            // If subarray size is less than 3, select the middle or the only element as pivot
            int pivotIndex = (low + high) / 2;
            swap(data[pivotIndex], data[high]);
            pivot = data[high].second;
        }
        else {
            // Randomly select three indices within the subarray
            int pivotIndex1 = std::rand() % (high - low + 1) + low;
            int pivotIndex2 = std::rand() % (high - low + 1) + low;
            int pivotIndex3 = std::rand() % (high - low + 1) + low;

            // Retrieve corresponding values
            long long pivot1 = data[pivotIndex1].second;
            long long pivot2 = data[pivotIndex2].second;
            long long pivot3 = data[pivotIndex3].second;

            // Find the median among the three random elements
            if ((pivot1 <= pivot2 && pivot2 <= pivot3) || (pivot3 <= pivot2 && pivot2 <= pivot1)) {
                pivot = pivot2;
                swap(data[pivotIndex2], data[high]);
            } else if ((pivot2 <= pivot1 && pivot1 <= pivot3) || (pivot3 <= pivot1 && pivot1 <= pivot2)) {
                pivot = pivot1;
                swap(data[pivotIndex1], data[high]);
            } else {
                pivot = pivot3;
                swap(data[pivotIndex3], data[high]);
            }
```

```
        }

    }

    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        if (data[j].second < pivot) {
            i++;
            swap(data[i], data[j]);
        }
    }
    swap(data[i + 1], data[high]);
    return (i + 1);
}
```

Notice that there is no difference in **swap** function compared to the same named function in the naive implementation of quicksort.

During the compilation of this code, our **main** function was as following.

```
int main() {

    std::vector<CityPair> data; // Vector of pairs to store CSV data
    csvToVector("dummyData.csv",data);

    const auto start{std::chrono::steady_clock::now()};
    part2_QuickSort(data, 0, data.size() - 1, 'l');
    const auto end{std::chrono::steady_clock::now()};
    const std::chrono::duration<long long, std::nano> elapsed_seconds{end - start};
    std::cout << "Time Taken: "<<elapsed_seconds.count() << " ns"<< std::endl;

    output(data, "out.csv");

return 0;
}
```

Here is the output of this code and content of our newly created **out.csv** file.

```
Time Taken:  3000 ns
```

	A	B
1	Pittston	1
2	ï»¿South Sioux City	2
3	Thompson	3
4	South Hill	4
5	Kashmar	5
6	Zhongli	6
7	Erfurt	7
8	San Lorenzo	8
9	Batna	9
10	Williamsburg	10
11	Rio Rancho	11
12	Prairie View	12
13	Fayetteville	13
14	Xi'an	14
15	Parana	15
16	Fair Lawn	16
17	Yibin	17
18	Commerce	18
19	Riverside	19
20	University	20

Figure 1.1: Content of out.csv after sorting

Notice that, there are different characters on the left side of "South Sioux City". This is because when the provided .csv files are opened in Visual Studio Code, it can be seen that they are encoded in "UTF-8 with BOM". However, we are trying to create a .csv file using our vector with encoding "UTF-8". Hence, when the program reads the first line of .csv file it adds "\357\273\77" in front of the first line. Since I do not want to corrupt the data, I did not change anything in this code to fix this problem.

By changing, the selection parameters among ’l’, ’r’, ’m’ and the filename parameter among "Population1.csv", "Population2.csv", "Population3.csv", "Population4.csv" the program has been run with all possible combinations. The results of those measurements in nanoseconds are given in the table below.

	Pupulation1	**Population2**	**Population3**	**Population4**
Last Element	297,263,600	4,439,495,200	2,562,865,600	8,624,800
Random Element	7,906,400	6,914,800	7,196,900	8,199,700
Median of 3	6,773,000	6,658,00	7,325,700	8,043,100

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

1.3. Hybrid Implementation of Quicksort and Insertion Sort

In this part, we are going to improve our code by adding a threshold parameter. Based on this threshold parameter, those functions are going to determine whether they are going to continue to operate the data in quicksort or insertion sort. In this part, we are going to examine the implementation details for each function. Just like in the previous part, we started to prepare an environment that makes us able to read the data and store them in a vector. Furthermore, since we need to swap the elements in further operations we used the same swap function. Notice that there is no difference between **split**, **csvToVector** and **swap** functions in this part compared to the same named functions in the previous part. Related code snippets can be seen below as they were in the previous part.

```
std::vector<std::string> split(const std::string &s, char delimiter) {
    std::vector<std::string> tokens;
    std::stringstream ss(s);
    std::string token;
    while (std::getline(ss, token, delimiter)) {
        tokens.push_back(token);
    }
    return tokens;
}
```

```
typedef std::pair<std::string, long long> CityPair;

void csvToVector(std::string filename,std::vector<CityPair> &data){
    std::ifstream file(filename); // CSV file name

    if (!file.is_open()) {
        std::cerr << "Error opening file!" << std::endl;
        return;
    }

    std::string line;
    while (std::getline(file, line)) {
        std::vector<std::string> row = split(line, ';'); // CSV data is
        ↪ semicolon-separated
        if (row.size() == 2) {
            std::string city = row[0];
            long long population = std::stoll(row[1]);
            CityPair cityData = std::make_pair(city, population);
            data.push_back(cityData);
        }
    }

    file.close();
}
```

```
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

The major change is done on the **quickSort** function. In addition to the previous quicksort function **threshold**, **firstArgumnet** and **Verbose** parameters are added. The threshold will be an integer that is determined by the user during the running of the program. Moreover, we changed the selection as firstArgument which is going to stand for our previous selection parameter. Based on firstArgumnet the quickSort function will determine which pivoting strategy is going to be applied to the data. Lastly, we added the verbose parameter in order to log the data. In other words, during the sorting operation if the user prefers to take the log of sub-collections and wants to see the progress, the user can use this option.

According to the obligations which are emphasized on the homework document some conditional cases are added into the quick sort function.

- **If threshold is equal to 1:** The naive quicksort strategy will be applied on the date. Since in our naive quicksort implementation, we only used the last element pivoting strategy, the related partition function and remaining quicksort functions for sub-collections will be called with 'l' parameter in which only the naive quicksort is applied. Notice that if the verbose is **true**, those sub-collections and the current pivot will be logged in **log.txt** file.

- **If the size of collection/sub-collection is less than or equal to threshold:** In this case, the current collection will be sorted by using insertion sort.

- **Else:** If the properties of the current collection/sub-collection do not fit any of these conditions the remaining quicksort functions and partition functions will be called regularly by the parameters that are taken from the user. In this case, we remain the pivoting strategy as it is. Notice that if the verbose is **true**, those sub-collections and the current pivot will be logged in **log.txt** file.

On the next page, the code snippet of the quicksort function can be seen.

```cpp
void quickSort(std::vector<CityPair>& data, int low, int high, int threshold,char
↪  firstArgument,bool verbose) {
    if (low < high) {

        if(threshold==1){
            int pi = partition(data, low, high, 'l');
            if(verbose)
            {
                std::ofstream myfile;
                myfile.open("log.txt", std::ios_base::app);
                myfile<<"Pivot: " << "("<<data[pi].first <<","<<
                ↪  data[pi].second<<")"<<" Array: "<<std::endl;

                for (int i=low; i<=high; i++) {
                myfile<<data[i].first <<";"<<  data[i].second<<std::endl;
                }
                myfile<<std::endl;
                myfile.close();
            }
            quickSort(data, low, pi - 1, threshold,'l',verbose);
            quickSort(data, pi + 1, high, threshold,'l',verbose);
        }
        else if (high - low + 1 <= threshold) {
            insertionSort(data, low, high,verbose);
        }

        else {
            int pi = partition(data, low, high, firstArgument);
            if(verbose)
            {
                std::ofstream myfile;
                myfile.open("log.txt", std::ios_base::app);
                myfile<<"Pivot: " << "("<<data[pi].first <<","<<
                ↪  data[pi].second<<")"<<" Array: "<<std::endl;

                for (int i=low; i<=high; i++) {
                myfile<<data[i].first <<";"<<  data[i].second<<std::endl;
                }
                myfile<<std::endl;
                myfile.close();
            }

            quickSort(data, low, pi - 1, threshold,firstArgument,verbose);
            quickSort(data, pi + 1, high, threshold,firstArgument,verbose);
        }
    }
}
```

Except for changing the name of the selection parameter to firstArgument no change is done on the **partition** function. Notice that the function works in the same logic which lies on the explanation in the previous part. The same code snippet for the partition function can be seen below.

```
int partition(std::vector<CityPair> &data, int low, int high, char firstArgument) {
    long long pivot;
    if (firstArgument=='l')
    {
        pivot = data[high].second;
    }

    else if(firstArgument=='r')
    {
        int randomNumber = std::rand() % (high - low + 1) + low;
        int pivotIndex= randomNumber;
        swap(data[pivotIndex], data[high]);
        pivot = data[high].second;
    }
    else if (firstArgument=='m')
    {
        if (high - low + 1 < 3) {
            // If subarray size is less than 3, select the middle or the only element as pivot
            int pivotIndex = (low + high) / 2;
            swap(data[pivotIndex], data[high]);
            pivot = data[high].second;
        }
        else {
            // Randomly select three indices within the subarray
            int pivotIndex1 = std::rand() % (high - low + 1) + low;
            int pivotIndex2 = std::rand() % (high - low + 1) + low;
            int pivotIndex3 = std::rand() % (high - low + 1) + low;

            // Retrieve corresponding values
            long long pivot1 = data[pivotIndex1].second;
            long long pivot2 = data[pivotIndex2].second;
            long long pivot3 = data[pivotIndex3].second;

            // Find the median among the three random elements
            if ((pivot1 <= pivot2 && pivot2 <= pivot3) || (pivot3 <= pivot2 && pivot2 <= pivot1)) {
                pivot = pivot2;
                swap(data[pivotIndex2], data[high]);
            } else if ((pivot2 <= pivot1 && pivot1 <= pivot3) || (pivot3 <= pivot1 && pivot1 <= pivot2)) {
                pivot = pivot1;
                swap(data[pivotIndex1], data[high]);
            } else {
```

```
                pivot = pivot3;
                swap(data[pivotIndex3], data[high]);
            }
        }

    }

    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        if (data[j].second < pivot) {
            i++;
            swap(data[i], data[j]);
        }
    }
    swap(data[i + 1], data[high]);
    return (i + 1);

}
```

Since we mentioned that we are going to sort the collection by insertion sort under some conditions, we implemented an insertion sort algorithm for the collection/sub-collections. Just like in quicksort function, if the user wants to log the stages of sorting insertion sort must be logged as well. Notice that since insertion sort does not select a pivot and does not create sub-collections, only the message is added to the log file.

```
void insertionSort(std::vector<CityPair>& vec, int left, int right, bool verbose) {
    for (int i = left + 1; i <= right; i++) {
        CityPair key = vec[i];
        int j = i - 1;

        while (j >= left && vec[j].second > key.second) {
            vec[j + 1] = vec[j];
            j = j - 1;
        }
        vec[j + 1] = key;
    }
    if(verbose){
        std::ofstream myfile;
        myfile.open("log.txt", std::ios_base::app);
        myfile<<"INSERTION SORT IS APPLIED"<<std::endl<<std::endl;
        myfile.close();
    }

}
```

The last function is going to be our **output** function. After sorting all of the collection, with the help of this function we are going to save our data in the output file with the desired name. This function takes two parameters. The **filename** parameter will be the name of our output file and the **data** vector is the data itself (i.e sorted data) that is desired to be kept in the output file. If there is an existing output file with the same name as the output file that is going to be created, this function removes the old one to avoid confusion. Related code snippet which belongs to this function is given below.

```
void output(std::vector<CityPair>& data, std::string outputFileName){
    remove(outputFileName.c_str());
    std::ofstream myfile;
    myfile.open(outputFileName, std::ios_base::app);
    for (const auto &city : data) {
    myfile<<city.first <<";"<<  city.second<<std::endl;
    }
    myfile<<std::endl;
    myfile.close();
}
```

In the main function, those arguments which are entered by the user are moved to the variables. Then, those variables are passed to the functions as parameters. However, those arguments are not directly assigned to those variables. We made a few type changes if it is needed. In this case **argv[1]**, **argv[2]**, **argv[3]**, **argv[4]** are assigned to **filename** (initial dataset file name), **firstArgument** (for selecting pivoting strategy), **threshold** and **outfilename** (in which sorted collection will be stored), respectively. Notice that since **verbose** is optional for users, **false** is assigned to this variable. If the related argument is entered by the user, this variable will be changed to **true**. The program starts with removing the old log file if it exists. Then, it keeps our data in a vector pair. Then, it sends the data with all the existing parameters to the quicksort function. Once the sorting is done, it sends the sorted collection to the output file with the desired name. Notice that the execution of quicksort function is measured by some variables. At the end, the difference between those two and other pieces of information are printed to the console as output.

During the running of the program **main** function was as following.

```cpp
int main(int argc, char* argv[]) {

    char firstArgument=*argv[2];
    int threshold= atoi(argv[3]);
    std::string filename= argv[1];
    std::string outfilename=argv[4];
    bool verbose=false;

    remove("log.txt");
    if (argc==6 && (*argv[5])=='v'){
         verbose=true;
    }

    std::vector<CityPair> data; // Vector of pairs to store CSV data
    csvToVector(filename, data);


    const auto start{std::chrono::steady_clock::now()};
    quickSort(data, 0, data.size() - 1,threshold, firstArgument,verbose);
    const auto end{std::chrono::steady_clock::now()};


    output(data, outfilename);

    const std::chrono::duration<long long,std::nano> elapsed_seconds{end - start};
    std::cout << "Time taken by QuickSort with pivot strategy \'"<<
    ↪  firstArgument<<"\' and threshold "
    <<threshold<<": " <<elapsed_seconds.count() << " ns"<< std::endl;
    return 0;
}
```

All the libraries that are needed for this program are given below.

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <fstream>
#include <sstream>
#include <chrono>
```

We compiled the program with the following shell command.

```
g++ -o QuickSort QuickSort.cpp
```

Once the compilation is done, we ran the program with the following shell command.

```
./QuickSort population4.csv r 5 out.csv
```

The console output for this program is given below.

```
Time taken by QuickSort with pivot strategy 'r'
and threshold 5:  7206700 ns
```

In order to fill the table that is given below, I used 'r' parameter for randomized pivot selecting strategy and different threshold (k) values. Those threshold values and corresponding function execution times in nanoseconds are given below. Notice that verbose is not entered for this measurement, because we only want to measure the execution of the function, logging might cause results to be too much in terms of time and inaccurate.

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

Threshold (k)	Population4
1	8,084,400
5	7,206,700
10	7,688,900
50	10,506,900
100	18,066,800
1,000	113,831,100
5,000	464,036,300
7,000	703,253,400
10,000	973,625,500
15,000	1,919,148,200

Notice that since no threshold value is specified for the measurements with respect to the pivoting strategy that we made in the table 1.1 the measurements are done by changing the main function in the code with the corresponding values. Since that code does not take any arguments after the compilation we run the code by `./QuickSort` shell command. Those measurements can also be done at the final version of the code if the threshold value is specified. Because in this stage since there is no threshold value our final code is not appropriate to make measurements with respect to different pivoting strategies. In order words, it does not run without any threshold value.

1.4. Discussion and Conclusion

Notice that our code is divided into three parts. The final version is not in the comment. If you want to see the development and run the related part, you have to put the other parts into the comment and leave the part of the code that you want to run. In all parts, there are their own main functions and the required functions. Hence take them as a comment or leave out them as a whole.

Normally quicksort algorithm has the following recurrence relation.

$$T(n) = 2T(n/2) + O(n)$$

However, in our code, we determined a threshold value (k) so that if the size of the collection is less than or equal to the threshold we are going to apply insertion sort. Since the insertion has O(n^2) time complexity, those sub-collections that satisfy this condition must have the same time complexity. Hence, in the end, the recurrence relation for our hybrid quicksort will be as follows on average.

$$T(n) = \begin{cases} O(n^2), & \text{if } n \leq k \\ T(n) = 2T(n/2) + O(n) & \text{otherwise} \end{cases}$$

Both quicksort and insertion sort are in-place sorting algorithms. Thus, they do not require extra memory space (i.e new array or vector). Nevertheless, since quicksort is a recursive function it takes space in the call stack. However, in our case this is negligible. Hence, our hybrid quicksort algorithm has O(1) space complexity. If we consider also those spaces space complexity will be between O(1) and O(logn). Since not all the calls for the sub-collections will be quicksort or insertion sort, the space complexity will be between those two.

Suppose that our threshold is too large in fact larger than the size of the collection. This means that the whole collection will be sorted by insertion sort based on our code. In this case, our time complexity will be O(n^2). However, if we choose a small value such as 2, only the collections that have size 2 or less insertion sort will be called. In this case, our time complexity will be too close to O(nlogn). Hence our time complexity for this code is between O(nlogn) and O(n^2). Our results show that for a collection that has too small sizes, insertion sort works better. This means that if the threshold is chosen as a large value hybrid quicksort does not work efficiently. However, if there is an appropriate threshold value, hybrid quicksort approach works slightly faster than the normal quicksort.