**Table of Contents**

[Quicksort: Theory and Experiments 2](#_Toc87548881)

[1. Pseudocode 2](#_Toc87548882)

[2. Growth Rate 4](#_Toc87548883)

[2.1 Experimental data 4](#_Toc87548884)

[2.2 Summary and Graph 7](#_Toc87548885)

[3. Algorithm Performance 9](#_Toc87548886)

[3.1 Pivot selection 9](#_Toc87548887)

[3.2 Further optimization 11](#_Toc87548888)

[3.3 Efficiency of Quicksort 14](#_Toc87548889)

[3.4 No O(n) Comparison-based Sorting Algorithm 15](#_Toc87548890)

[3.5 Beyond Comparison-based Sorting Algorithm 15](#_Toc87548891)

[References 17](#_Toc87548892)

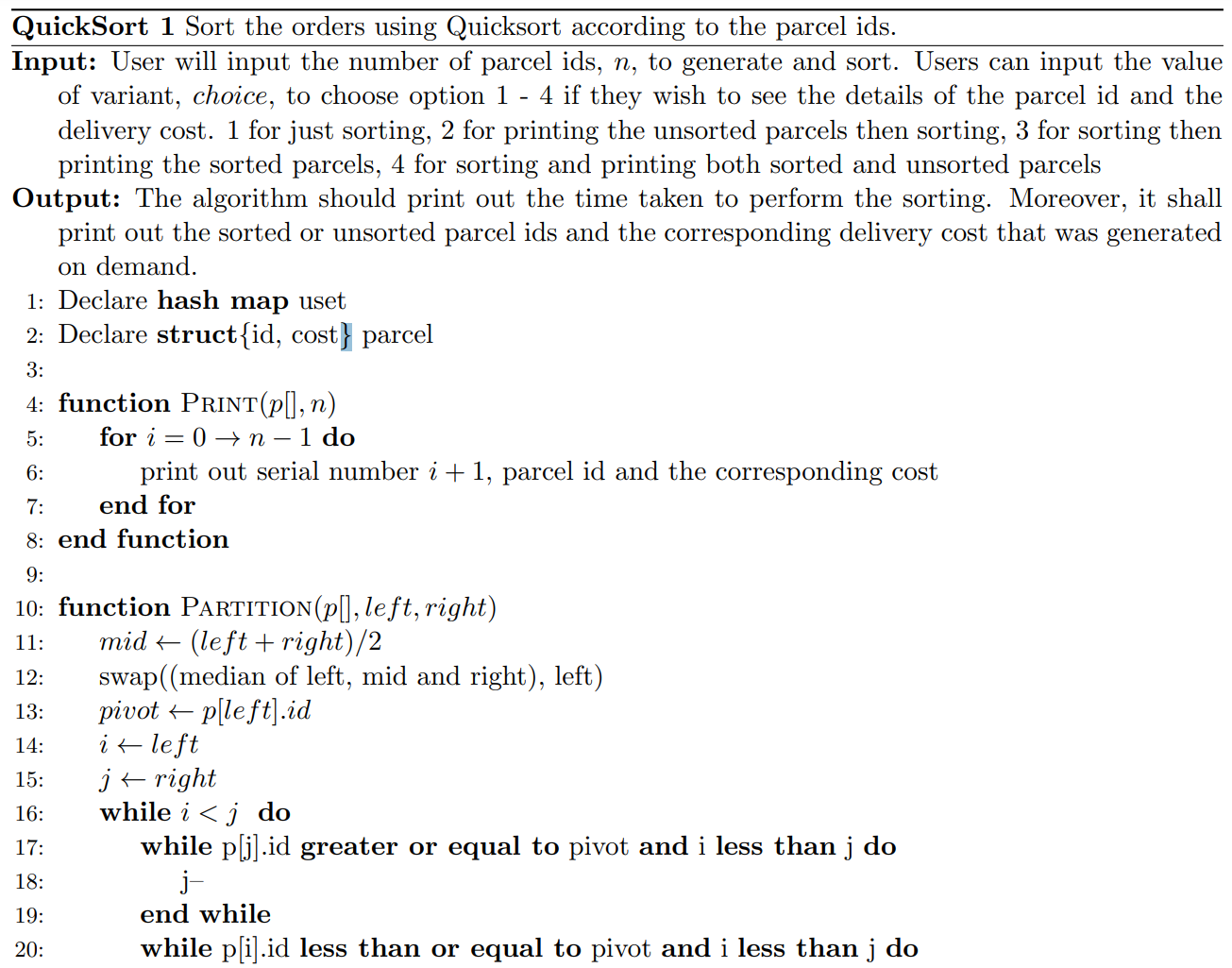
[Appendix 1: Source Code 18](#_Toc87548893)

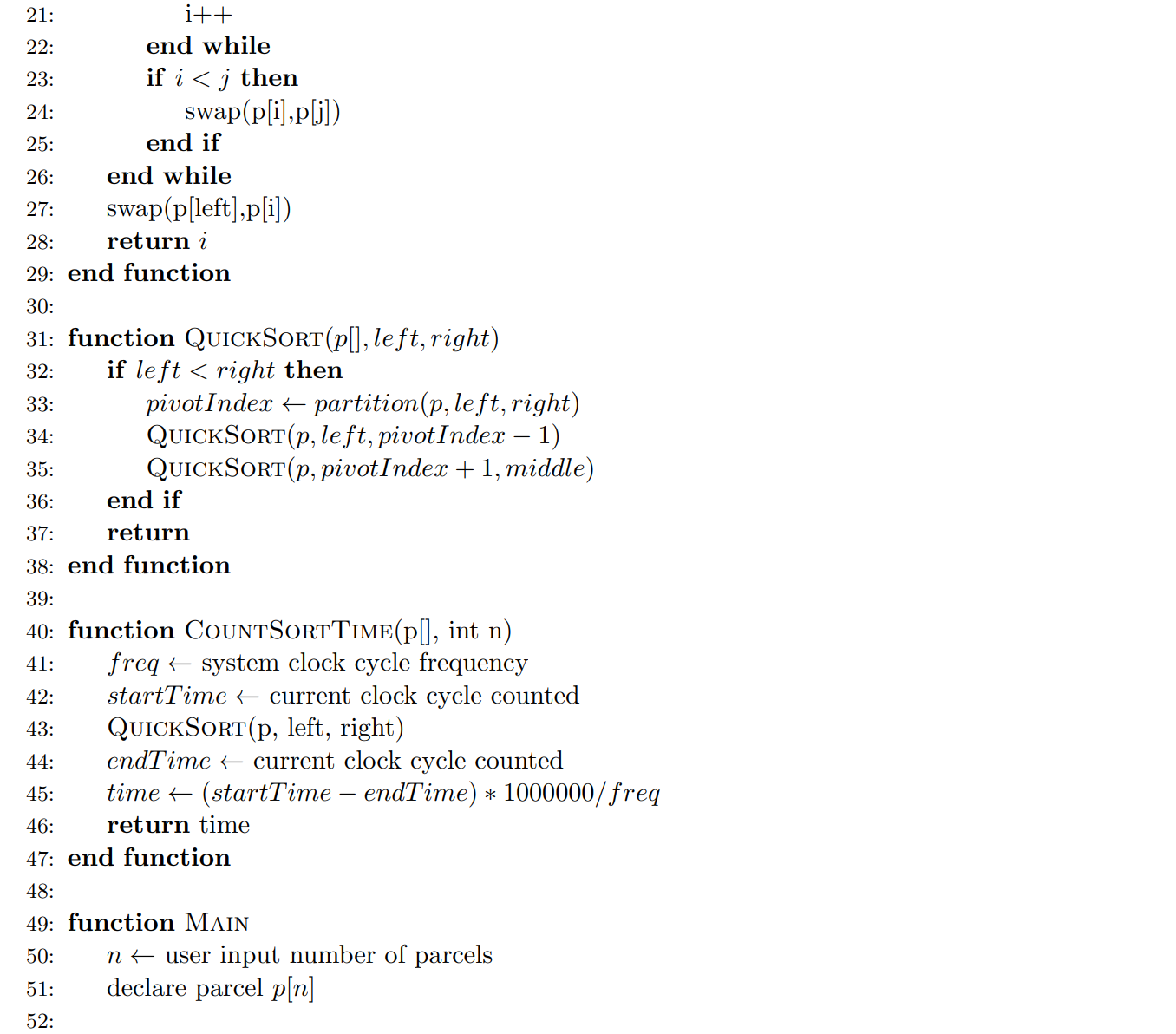
[Appendix 2: Marking Rubrics 22](#_Toc87548894)

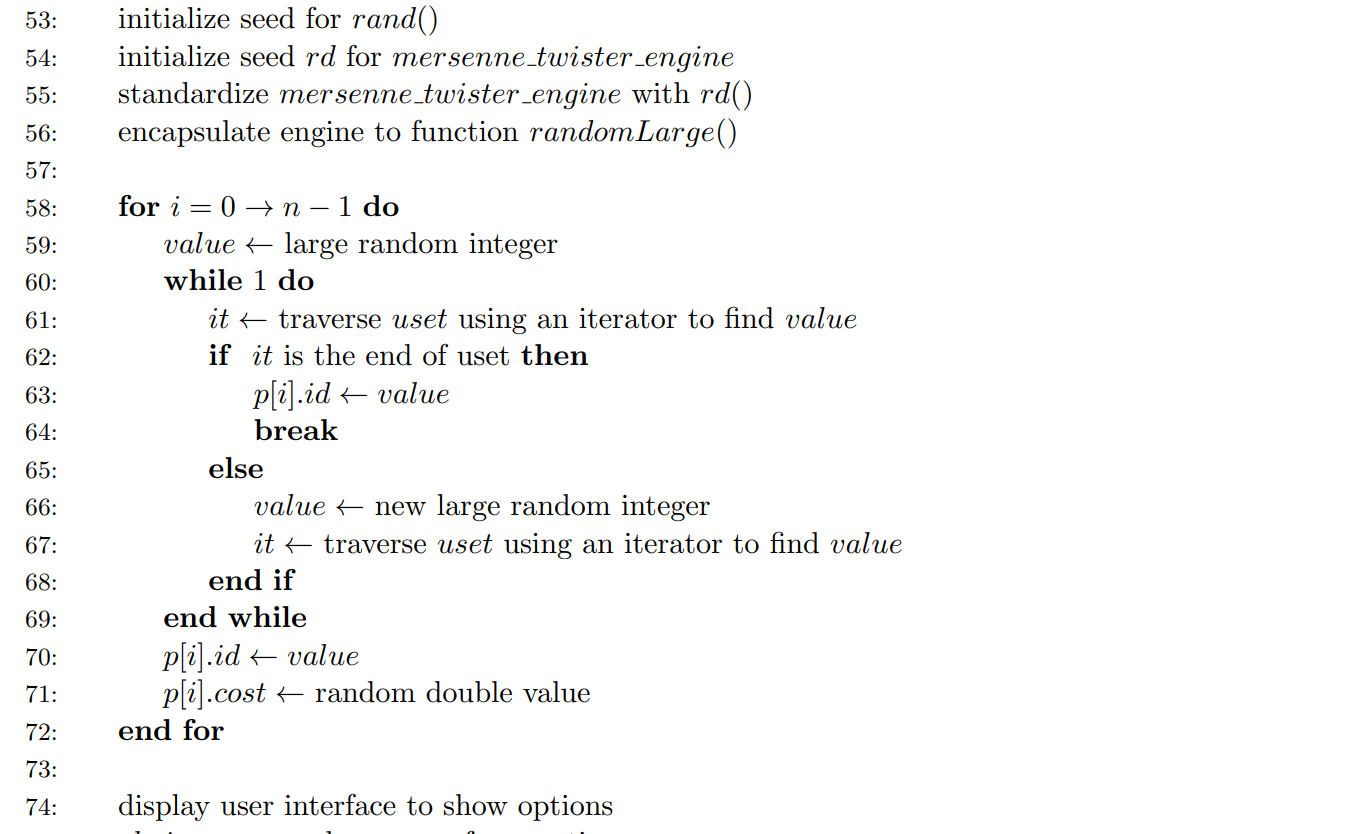
# Quicksort: Theory and Experiments

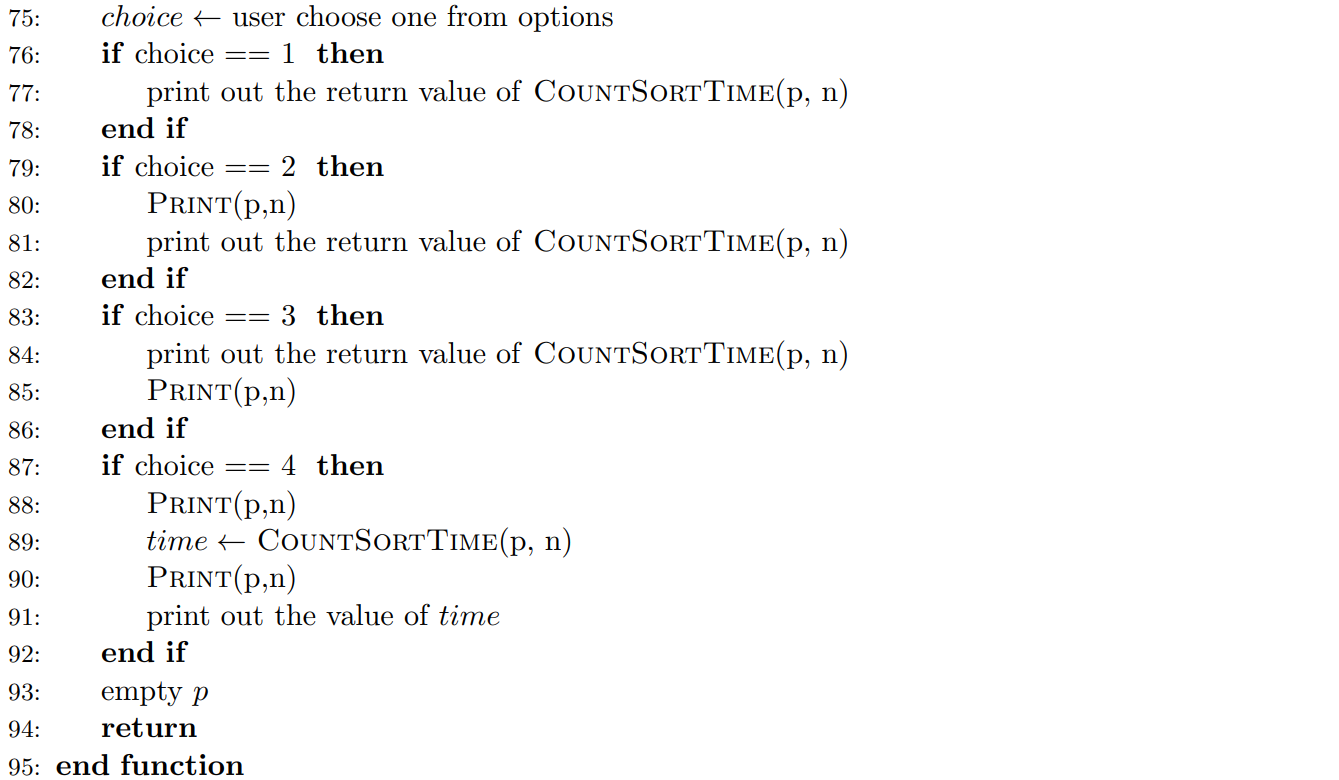
## Pseudocode

Pseudocode is an effective way to abstracts away the syntax to let us focus on solving the problem in front of us instead of getting bogged down in the exact syntax language. Moreover, it allows us to work on pure programming logic which provides us a chance to simply write what it is in plain English. Therefore, we start with pseudocode.









## Growth Rate

Normally, growth rate refers to the how the scale of the algorithm's time complexity and space complexity increase as the scale grows. In addition to predicting the performance of the algorithm, analyzing the growth rate helps to classify problems as well as algorithms by difficulty. This is very useful when we compare different algorithms accordingly.

Under normal circumstances, we can analyze the growth rate of the algorithm through two methods: empirical analysis and theoretical analysis (Vaz, Shah, Sawhney & Deolekar, 2017). Here we apply the empirical analysis to find the growth rate of the algorithm.

### Experimental data

To test the growth rate of the program using empirical analysis method, we run the program with the following n values: 1000, 3000, 5000, 8000, 100000, 15000, 25000, 35000, 51200, 66000, 86400, 100000, 125000, 150000, 180000, 200000, 250000, 300000, 4000000, 500000, 600000, 700000, 800000, 900000, 1000000.

We all know that the result of one time does not explain anything, and there may be deviations. In order to ensure the accuracy of the data, we adopt the method of taking the average of multiple measurements. For each of them, we will run for ten times and take the average value as the final value.

Here is the table for the measurement:

1. Experiment Result From 1000 - 25000

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1000** | **3000** | **5000** | **8000** | **10000** | **15000** | **25000** |
| **Experiment1** | 93.5 | 351.1 | 598.2 | 896.3 | 995.3 | 1505.0 | 2989.5 |
| **Experiment2** | 85.5 | 349.2 | 537.6 | 777.5 | 1128.6 | 1496.6 | 2987.6 |
| **Experiment3** | 88.2 | 288.3 | 541.1 | 798.1 | 995.7 | 1627.4 | 3550.2 |
| **Experiment4** | 96.1 | 312.8 | 568.3 | 911.5 | 1207.3 | 1396.9 | 3306.7 |
| **Experiment5** | 99.6 | 311.9 | 538.1 | 882.6 | 1134.9 | 1456.2 | 3107.6 |
| **Experiment6** | 95.7 | 271.7 | 628.3 | 788.8 | 1085.2 | 1640.7 | 2860.8 |
| **Experiment7** | 87.2 | 289.5 | 541.4 | 852.4 | 1198.3 | 1505.3 | 3300.4 |
| **Experiment8** | 89.1 | 309.9 | 562.4 | 889.1 | 1067.5 | 1550.4 | 3421.5 |
| **Experiment9** | 84.5 | 301.2 | 589.7 | 873.6 | 952.2 | 1589.0 | 3523.4 |
| **Experiment10** | 86.2 | 303.6 | 601.2 | 885.4 | 987.4 | 998.6 | 3601.9 |
| **Average** | 90.6 | 308.9 | 570.6 | 855.5 | 1075.2 | 1476.6 | 3265.0 |

1. Experiment Result From 35000 -150000

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **35000** | **51200** | **66000** | **86400** | **100000** | **125000** | **150000** |
| **Experiment1** | 4305.9 | 5592.3 | 7487.2 | 10028.9 | 13214.8 | 16932.8 | 19908.0 |
| **Experiment2** | 4290.6 | 5166.1 | 6689.7 | 9730.2 | 12551.0 | 16301.4 | 18376.2 |
| **Experiment3** | 4019.6 | 4984.9 | 6624.4 | 10784.8 | 12790.9 | 15925.9 | 20291.9 |
| **Experiment4** | 4400.7 | 5380.9 | 6664.9 | 10069.0 | 12761.1 | 16476.8 | 19234.9 |
| **Experiment5** | 4002.5 | 4810.7 | 6655.9 | 9645.5 | 13457.1 | 16049.6 | 20474.8 |
| **Experiment6** | 4104.4 | 5243.0 | 6677.8 | 10882.4 | 12999.3 | 16861.1 | 19275.0 |
| **Experiment7** | 4003.3 | 5067.7 | 6638.6 | 9966.0 | 12852.6 | 15140.6 | 20116.7 |
| **Experiment8** | 4307.1 | 5101.7 | 6674.1 | 10662.9 | 12633.1 | 15678.9 | 19507.6 |
| **Experiment9** | 4461.1 | 5376.7 | 6622.7 | 10962.3 | 12845.8 | 15154.2 | 18954.7 |
| **Experiment10** | 4040.4 | 4976.7 | 6611.8 | 10808.6 | 12562.0 | 15003.0 | 19960.8 |
| **Average** | 4193.6 | 5170.1 | 6734.7 | 10354.1 | 12866.8 | 15952.4 | 19610.1 |

1. Experiment Result From 180000 -600000

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **180000** | **200000** | **250000** | **300000** | **4000000** | **500000** | **600000** |
| **Experiment1** | 23917.2 | 26060.3 | 32352.6 | 38931.7 | 51491.6 | 66087.5 | 80563.8 |
| **Experiment2** | 21946.7 | 27172.8 | 29758.0 | 37532.6 | 48837.0 | 66646.8 | 78239.4 |
| **Experiment3** | 21186.3 | 25759.9 | 30055.8 | 36384.8 | 50522.0 | 65820.0 | 80201.9 |
| **Experiment4** | 21580.1 | 26417.3 | 30379.3 | 36802.6 | 50023.3 | 63413.9 | 74872.5 |
| **Experiment5** | 23456.6 | 26765.9 | 29505.6 | 36721.7 | 52330.6 | 65212.4 | 78974.1 |
| **Experiment6** | 23561.8 | 27482.3 | 29859.6 | 39207.1 | 48657.6 | 72475.8 | 77629.2 |
| **Experiment7** | 23311.9 | 26960.4 | 30555.2 | 35618.3 | 53729.0 | 68441.6 | 77005.6 |
| **Experiment8** | 21331.6 | 26020.5 | 31193.8 | 36113.0 | 50215.1 | 63275.0 | 81014.4 |
| **Experiment9** | 22532.6 | 27263.7 | 30975.6 | 38039.2 | 49329.2 | 72554.4 | 74982.7 |
| **Experiment10** | 23383.9 | 25801.6 | 31620.9 | 36535.0 | 49896.3 | 64280.8 | 82908.5 |
| **Average** | 22620.9 | 26570.5 | 30625.6 | 37188.6 | 50503.2 | 66820.8 | 78639.2 |

1. Experiment Result From 700000 -1000000

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **700000** | **800000** | **900000** | **1000000** |  |  |  |
| **Experiment1** | 92960.4 | 106881.0 | 118951.0 | 135267.0 |  |  |  |
| **Experiment2** | 95680.8 | 92260.1 | 111529.1 | 127008.5 |  |  |  |
| **Experiment3** | 94077.4 | 92694.6 | 112814.4 | 126985.8 |  |  |  |
| **Experiment4** | 85720.6 | 105417.3 | 112653.3 | 129414.8 |  |  |  |
| **Experiment5** | 92945.2 | 96946.4 | 110922.3 | 133788.7 |  |  |  |
| **Experiment6** | 87461.9 | 100097.9 | 116653.5 | 138450.3 |  |  |  |
| **Experiment7** | 93838.2 | 91980.6 | 117413.5 | 127723.6 |  |  |  |
| **Experiment8** | 89178.0 | 95847.4 | 115637.9 | 133766.2 |  |  |  |
| **Experiment9** | 90659.1 | 96326.0 | 113625.5 | 140870.8 |  |  |  |
| **Experiment10** | 87643.1 | 105245.6 | 118134.6 | 126903.4 |  |  |  |
| **Average** | 91016.5 | 98369.7 | 114833.5 | 132017.9 |  |  |  |

### Summary and Graph

Although the rate of growth varies with the configuration of the machine, the trend is generally the same. To better illustrate it, we plot a graph to show the growth rate:

1. Relationship between running time and array size

From the class, we already know that the best case the quicksort it , while the worst case is. So, the range where we fit this function is probably from t.

To find the trend of the average cost, we fit the above function with the following polynomial:

Obviously, will play an important role in our current data range, so we may first start with growth rate . I try to draw the graph of , , respectively, but they are all larger than the current data. After several attempts, I got the most suitable fitting function which is above our average cost in all values:

1. Average cost and 0.04nlogn

Similarly, we also find another function which is below our average cost in all values.

1. Average cost and 0.01nlogn

By definition, we know that the growth rate of this algorithm is

## Algorithm Performance

As an intensively studied problem in the field of computer science, the sorting problem has attracted a large number of researchers to focus on it. Moreover, sorting algorithms have a significant impact on performance on complex computing tasks. Even 5% performance optimization will bring a very obviously improvement. Therefore, it is worthwhile to take the effort to solve this problem.

There is no doubt that optimizing the quicksort algorithm is very important. As an intern of quicksort Inc., I was supposed to develop a quicksort algorithm with higher performance and I will discuss how to optimize quicksort in this paper.

### Pivot selection

Adopting the idea of divide and conquer, quicksort splits the big into small ones and splits the small into smaller ones. In simple terms, the principle of quicksort is to select a pivot, divide the original array into two parts by comparing each element in the original array with pivot, and repeat this process continuously. After sorting all the small arrays, the final result is the already sorted.

Therefore, the choose of pivot is very important for the efficiency of quicksort algorithm. The pivot strategy we used is the median of three, here I will illustrate why I choose this method by giving a detailed explanation to several commonly used pivot selection methods.

However, if I use different pivots, it may faster algorithm in linear but not exponential. The upper bound is .

1. **Choose First**

This method is the simplest, just return the subscript of the first element of the sub-array, which is implemented below.

int partition(int\* arr, int left, int right)  
 int pivot = arr[left];

The only reason for choosing the *first* as pivot is because this method is relatively simple and easy to implement. However, it shall be noted that it is very likely to deteriorate to the worst case while using this pivot.

1. **Choose Last**

This method is the same as the previous method, but we selected the *last* element.

int partition(int\* arr, int left, int right)  
 int pivot = arr[right];

1. **Randomized quicksort**

Here, we use a random function to randomly generate a number in the array, the variable *left* is the left boundary of the current array, and the variable *right* is the right interface. We use rand()%(right-left)+left) to randomly select.

int partition(int\* arr, int left, int right)  
 swap(arr[rand() % (right - left) + left], arr[left]);  
 int pivot = arr[right];

The pivot generated by **randomization** may help to solve this problem, but at the same time, it should be noted that the random numbers generated by C++ are pseudo-random numbers generated according to algorithms and random seeds, and most of them have **defects**.

For example, the random numbers generated by the *rand()* function in a very short time are the same. In addition, in order to ensure the randomness of random numbers, these algorithms often require a lot of calculations, which will **consume a lot of resources** and **affect our sorting speed**.

1. **Median of three partitioning**

Select the median of the first, last and middle element.

int partition(int\* arr, int left, int right)  
 int mid = (left+right)/2  
 int pivot =  
 min(min(max(left,mid),max(mid,right)),max(right,left))

Choosing the first or last one may be due to the fact that the array is close to being sorted and **deteriorated** to an algorithm. Randomization generation can also take a lot of time, and the speed of the algorithm is **uncertain**. Taking the median of the three numbers may be able to reduce this situation. Usually, we consider this selection way to be the **best** choice. **So, this is the method we choose**.

There are other improved versions of median of three, such as **median of five** and **median of seven**, but they are essentially the same.

### Further optimization

Although the current growth rate of our algorithm is almost satisfactory, we need to note that this is not the best strategy when the range of the array is too large or too small. One way of better implementation may be the STL *sort* function (Faujdar & Ghrera, 2016), which is included in the C++ header file *<****algorithm.h****>*.

The STL sort function not just ordinary quicksort in addition to **optimizing** ordinary quicksort, it also combines insertion sort and heap sort. According to different quantity levels and different situations, the appropriate sorting method can be **automatically selected**. When the amount of data is large, it will try to use the method of quicksort **partition and recursion** at first. Once the amount of data after partition is less than a certain threshold, to **avoid excessive extra load** caused by recursive calls, **insertion sort** will be used instead (SONG, FU, ZHANG, PENG & LIANG, 2010). If the recursion level is **too deep**, there is a tendency for the worst case to occur, and **heap sort** will be used instead.

So here are other better sorting techniques in different situation.

* 1. **switch to insertion sort in small array**

Fast sorting requires constant recursion. When the array is very small, we can use insertion sorting. Insertion sorting is to insert one element at a time on the basis of an already ordered small sequence. When the length of the sequence to be sorted is between 5 and 20, the use of insertion sort at this time can avoid some harmful degeneration situations (Shaffer, 2012). It might be faster to use insertion sort in this case.

* 1. **switch to heap sort when the recursion is too deep**

Heap sorting mainly uses a data structure called a heap, also known as a binary heap. This is a sorting strategy based on comparison. The process of putting elements into the heap data structure is called heapify. Through the adjustment of the heap, we complete the final sorting process. It is often used when the amount of data is extremely large (Li, Chen & Wang, 2017).

In addition, we also have other ways to optimize. Here is what we can do to further improve the efficiency of the algorithm.

* 1. **multiple-pivot quicksort**

Since 70s of the last century, some researchers have been committed to implementing double-pivot and three-pivot quicksort algorithms. The paper published by Kushagra *et al.* also talk about the probability of multi pivot quicksort (Kushagra, López-Ortiz, Qiao & Munro, 2013). This quicksort algorithm uses n pivots to divide the original array into n+1 arrays, which is a further divide and conquer algorithm. However, according to the result of Budiman (Budiman, Zamzami & Rachmawati, 2017) , the performance of the multi pivots algorithm have a great relationship with cache performance and it works best when the number of pivots is three in most case.

* 1. **Gather elements with the same value**

After a partition is over, the elements equal to pivot can be grouped together, and when the next division continues, there is no need to divide the elements equal to pivot again. In this way, by clustering the elements equal to pivot, the **number of iterations** can be **reduced**, and the efficiency will be improved a lot (Wild, 2018).

* 1. **find better pivot strategy**

As we discussed above, a better pivot selection strategy has a decisive influence on the performance of quicksort. It is necessary to choose a better pivot selection scheme to improve the quicksort algorithm.

Apart from these commonly used pivot selection methods mentions, we also have other pivot selection method that might be better. In some special cases, for example, when the most elements of the array are sorted except some of them, we can make targeted improvements to pivot selection strategy. Especially in the field of engineering calculation or graphics calculation, a lot of repetitive work is often required.

* 1. **Tail recursion**

If a recursive function calls itself at the end of the function, at this time, wee can overwrite the current record instead of creating a new one, thereby improving efficiency. For example, the capacity of our code stack is limited. The conventional method can only sort an array of about 50,000. When we use tail recursion, we can effectively increase the maximum sort, double or even triple.

* 1. **multithreading and multiprocessing**

Hardware such as memory, CPU, etc. determine the processing speed. The unit that we allocate resources is the thread. Therefore, if we open multiple processes, more resources will be allocated and tasks will be completed faster. The main effect of multithreading is to increase the number of concurrencies.

In addition, we also use multi-threading to improve resource efficiency. Multiple tasks take turns using the CPU.

### Efficiency of Quicksort

The complexity of our quicksort is approximately , which has a certain relationship with the implementation of the algorithm and the configuration of the machine (Jadoon, 2019).

However, no matter what host machine it is running on, the growth rate would always be the same. In this part, we will analysis the complexity of quicksort in best average and worst case to show why my algorithm appears in that way.

For best case, each time partition can separate it to half, so from n to 1 we need do times. But for each level, we shall traverse all the elements.

For average case, we assume that partition can happen in n position each with probability .

Therefore, the complexity of average case is

In the worst case, the array is sorted or the elements in it are all equal, quicksort degenerates into bubble sort and we know the time complexity of bubble sort is because each element in the array will compare with other elements.

### No O(n) Comparison-based Sorting Algorithm

Suppose we have an array to be sorted which has the **size of n**. If we want to order it, we need to access each element at least once to know all the information, but we can't use the additional resource on the constraints of **comparison-based** sorting algorithm (Bustos, Pedreira & Brisaboa, 2008). So, we need more actions to achieve the goal of sorting the array.

We carry out the comparison of sorting algorithms in pairs. And we can abstract it into a decision tree. we compare the left child node and the right child node in each node. For an array of length n, there are  **combinations of elements**. The result of sorted results is one of them. So, we have leaves for the decision tree.

Every time we do a sort, we eliminate at most half of the possibilities. We divide all cases where the left subtree is larger than the right subtree into one pile, and divide all cases where the right subtree is larger than the left subtree into another pile. At the beginning we have possibilities, and at the end we only have one possibility, which is the sorted array that we want.

Assume we get the final result after **k comparision**:

Therefore,

Because:

For :

### Beyond Comparison-based Sorting Algorithm

For **comparison-based** sorting algorithm, it is possible to improve it a bit. For example, we might improve its time complexity from 0.05nlogn to 0.03nlogn. However, it is impossible to improve it from unless we use additional resources.

It is for sure that we cannot create an O(n) comparison-based sorting algorithm, but if we use additional spaces, we do have better techniques. Here are some of them.

* 1. **radix sort**

We divide the maximum value in the array according to the number of digits, and then sort it by units, then by 10, compare each bit, and so on to make it sorted. When fetching data, according to the queue’s rule: first in, first out. The time complexity would be .

* 1. **count sort**

the step of count sort is we create a auxiliary array to store elements in the auxiliary array, traverse the elements in the original array, use the elements in the original array as the index of the count array, and use the number of occurrences of the elements in the original array as the count array element value.

* 1. **bucket sort**

As an extended version of counting and sorting, we first group the sorted numbers into several different buckets. Then we use the mapping function to calculate the corresponding mapping value of the elements in the array. When needed, we directly access it through the subscript. It needs O(n) times operation to do the mapping. However, once the mapping is done in advance, the speed can reach when searching (Faujdar & Saraswat, 2017).

**References**

Bahig, H. (2019). Complexity analysis and performance of double hashing sort algorithm. *Journal Of The Egyptian Mathematical Society*, *27*(1). doi: 10.1186/s42787-019-0004-2

Budiman, M., Zamzami, E., & Rachmawati, D. (2017). Multi-Pivot Quicksort: an Experiment with Single, Dual, Triple, Quad, and Penta-Pivot Quicksort Algorithms in Python. *IOP Conference Series: Materials Science And Engineering*, *180*, 012051. doi: 10.1088/1757-899x/180/1/012051

Bustos, B., Pedreira, O., & Brisaboa, N. (2008). A Dynamic Pivot Selection Technique for Similarity Search. *First International Workshop On Similarity Search And Applications (Sisap 2008)*. doi: 10.1109/sisap.2008.12

Faujdar, N., & Ghrera, S. (2016). Performance Evaluation of Parallel Count Sort using GPU Computing with CUDA. *Indian Journal Of Science And Technology*, *9*(15). doi: 10.17485/ijst/2016/v9i15/80080

Faujdar, N., & Saraswat, S. (2017). The detailed experimental analysis of bucket sort. *2017 7Th International Conference On Cloud Computing, Data Science & Engineering - Confluence*. doi: 10.1109/confluence.2017.7943114

Jadoon. (2019). Design and Analysis of Optimized Stooge Sort Algorithm. *International Journal Of Innovative Technology And Exploring Engineering*, *8*(12), 1669-1673. doi: 10.35940/ijitee.l3167.1081219

Kushagra, S., López-Ortiz, A., Qiao, A., & Munro, J. (2013). Multi-Pivot Quicksort: Theory and Experiments. *2014 Proceedings Of The Sixteenth Workshop On Algorithm Engineering And Experiments (ALENEX)*, 47-60. doi: 10.1137/1.9781611973198.6

Li, H., Chen, P., & Wang, Y. (2017). Heap Sorting Based on Array Sorting. *Journal Of Computer And Communications*, *05*(12), 57-62. doi: 10.4236/jcc.2017.512006

Shaffer, C. (2012). *Data Structures and Algorithm Analysis in C++*. Dover Publications.

SONG, H., FU, Y., ZHANG, L., PENG, H., & LIANG, H. (2010). Multi-thread quicksort algorithm based on partitioning. *Journal Of Computer Applications*, *30*(9), 2374-2378. doi: 10.3724/sp.j.1087.2010.02374

Vaz, R., Shah, V., Sawhney, A., & Deolekar, R. (2017). Automated Big-O analysis of algorithms. *2017 International Conference On Nascent Technologies In Engineering (ICNTE)*. doi: 10.1109/icnte.2017.7947882

Wild, S. (2018). Dual-pivot and beyond: The potential of multiway partitioning in quicksort. *It - Information Technology*, *60*(3), 173-177. doi: 10.1515/itit-2018-0012

**Appendix1**

**Source Code**

#**include** <windows.h>  
#**include** <bits/stdc++.h>  
#**define** random(x) rand() % (x)  
**using** **namespace** std;  
unordered\_set<**long**> uset;  
  
**struct** **parcel**  
{  
 **int** id;  
 **double** cost;  
};  
  
**void** **print**(parcel \*p, **int** num)  
{  
 **for** (**int** i = 0; i < num; i++)  
 cout << "[" << i + 1 << "] "  
 << "\tParcel ID:" << p[i].id << "\tCost:"  
 << "RM"  
 << setiosflags(ios::fixed) << setprecision(2) << p[i].cost << endl;  
}  
  
**int** **partition**(struct parcel p[], **int** left, **int** right)  
{  
 swap(p[rand() % (right - left) + left], p[left]);  
 **int** pivot = p[left].id;  
 **int** i = left;  
 **int** j = right;  
 **while** (i < j)  
 {  
 **while** (p[j].id >= pivot && i < j)  
 j--;  
 **while** (p[i].id <= pivot && i < j)  
 i++;  
 **if** (i < j)  
 swap(p[i], p[j]);  
 }  
 swap(p[left], p[i]);  
 **return** i;  
}  
  
**void** **qSort**(struct parcel p[], **int** left, **int** right)  
{  
 **if** (left < right)  
 {  
 **int** pivotIndex = partition(p, left, right);  
 qSort(p, left, pivotIndex - 1);  
 qSort(p, pivotIndex + 1, right);  
 }  
}  
  
**double** **countSortTime**(struct parcel p[], **int** num)  
{  
 LARGE\_INTEGER freq;  
 QueryPerformanceFrequency(&freq);  
 LARGE\_INTEGER startTime;  
 LARGE\_INTEGER endTime;  
 QueryPerformanceCounter(&startTime);  
  
 qSort(p, 0, num - 1);  
  
 QueryPerformanceCounter(&endTime);  
 **double** runTime = (endTime.QuadPart - startTime.QuadPart) \* 1000000.0 / freq.QuadPart;  
 **return** runTime;  
}  
  
**int** **main**()  
{  
 **int** num, choice;  
 cout << "Please input the number of parcel ids: ";  
 cin >> num;  
  
 parcel \***const** p = **new** parcel[num]();  
  
 srand((**unsigned**)time(NULL));  
 std::random\_device rd; //Will be used to obtain a seed for the random number engine  
 std::mt19937 **gen**(rd()); //Standard mersenne\_twister\_engine seeded with rd()  
 std::uniform\_int\_distribution<> randomLarge(1, num \* 30);  
 //https://en.cppreference.com/w/cpp/numeric/random/uniform\_int\_distribution  
  
 **for** (**int** i = 0; i < num; i++)  
 {  
 **long** value = randomLarge(gen);  
 **while** (1)  
 {  
 **auto** it = uset.find(value);  
 **if** (it == uset.end())  
 {  
 uset.insert(value);  
 **break**;  
 }  
 **else**  
 {  
 value = randomLarge(gen);  
 it = uset.find(value);  
 }  
 }  
 p[i].id = value;  
 p[i].cost = random(9999) + random(100) / 100.0;  
 }  
  
 cout << "\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* QuickSolve Inc.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl  
 << "\t\t\* \*" << endl  
 << "\t\t\* \*" << endl  
 << "\t\t\* 1. Run silently \*" << endl  
 << "\t\t\* 2. Run and display unsorted list of parcels id and cost \*" << endl  
 << "\t\t\* 3. Run and display sorted list of parcels id and cost \*" << endl  
 << "\t\t\* 4. Run and display both unsorted and sorted list of parcels id and cost \*" << endl  
 << "\t\t\* \*" << endl  
 << "\t\t\* \*" << endl  
 << "\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl  
 << "Please input your choice: ";  
 cin >> choice;  
 **switch** (choice)  
 {  
 **case** 1:  
 cout << "The running time of algorithm is " << countSortTime(p, num) << " ns." << endl;  
 **break**;  
 **case** 2:  
 print(p, num);  
 cout << "The running time of algorithm is " << countSortTime(p, num) << " ns." << endl;  
 **break**;  
 **case** 3:  
 cout << "The running time of algorithm is " << countSortTime(p, num) << " ns." << endl;  
 print(p, num);  
 **break**;  
 **case** 4:  
 print(p, num);  
 **int** time = countSortTime(p, num);  
 print(p, num);  
 cout << "The running time of algorithm is " << time << " ns." << endl;  
 **break**;  
 }  
 **delete**[] p;  
 **return** 0;  
}