**BCoT-Based Smart Manufacturing: An Enhanced Precise Measurement Management System**

***Abstract***

Block chain with its transparent, decentralized and secured characteristics have surfaced as a futuristic technology for a surplus of advanced industrial applications. Block chain of Things (BCoT) shows the merging of Block chain and IoT. The developments in multi-virtual Sensor IoT, homogeneous and heterogeneous multi-system information fusion for BCoT, industrial applications of BCoT has transformed the way digital world work. Ever since the smart devices were introduced, the world has evolved and progressed by making the entire world more dynamic by bringing people, technology, and machines closer to one another.

The smart manufacturing sector is another developing field where BCoT can spread its contribution. Smart manufacturing can be indicated as “organizing automated machines which can perceptively perform tasks efficiently.” This sector is basically based on [IoT-enabled technologies](https://www.oodlestechnologies.com/iot-application-development-services/), cloud manufacturing, and service-oriented manufacturing.

Moreover The IoT and its counterpart, the [Industrial Internet of Things (IIoT)](https://en.wikipedia.org/wiki/Industrial_Internet_of_Things) are enhancing sensor usage and contributing to smart manufacturing. There are different kinds of sensors such as Temperature sensors, Humidity sensors, pressure sensor, Accelerometers, Gyroscope sensors, gas sensors, infrared sensors, optical sensors, the mythings smart sensors, proximity sensors etc.

In this paper, we will discuss about sensors, proximity sensors in details, customer and product movement, reaction of customers towards sensors and smart technologies based on BCoT, a precise measurement system using quick sort usable in smart manufacturing and future challenges and possibilities.

***Key words:*** BCoT, measurement system using quick sort, Proximity sensor, Smart manufacturing and Sensors.

***Introduction:***

BCoT applied to various manufacturing applications and helps to create an automatic, decentralised smart manufacturing system with a high level of efficiency and productivity which stands as a savior for diverse business sectors and industries that are looking to maintain data privacy, decentralization, improvement and accessibility. The use of Block chain because of having its characteristics widely increased at a breakneck speed. Blockchain reference architecture for smart manufacturing is well-known to the digital world on applying the blockchain technology to various applications of the smart factory and smart supply chain.

Blockchain-driven secure connectivity, fundamental characteristics of IoT, the BCoT Internet Industry 5.0 of Internet of Things is revolutionizing Industry 5.0 to highlighted one. Blockchain, implementation of block chain in IoT, using sensors of IoT in different devices address a lot of challenges to overcome. As we see many limitations still exiting. Moreover, people started being digital starting from rural area to the center of cities. Introduction of BCoT and sensors in digital devices are becoming indispensable part of the digital devices and the core attention of the research area to expand and improve more to make life easier.

Sensors are used in our homes and workplaces being embedded in smart phones and an integral part of the Internet of Things (IoT), in different formal and informal infrastructures where we use digital technologies widely for security or to detect or collect data from any harbingers for example in industries, shopping centers, hospitals.

I would like to mention a brief idea on the working principle of sensors. Sensors detect and respond to fluctuations in an environment where inputs can come from a variability of sources such as pressure light, motion and temperature. These devices are capable to output information on the basis of what they detect and if these are connected to a network, these can share data with other connected devices and management systems. IoT sensors are used widely thus this has a lot of scope of improvements and to improve operational efficiency, to reduce costing, to enhance workers safety, to improve software and hardware development of the IoT sensors devices to spread more use of them and to make use of them easier and more effective, we need to focus on different factors.

Proximity sensors which are applicable in industrial sectors mostly. In this paper, I would like to focus on the improvement of technology which would be more relevant in large hypermarkets, niche clothing stores, food supermarkets, shoes supermarkets, furniture stores or in other centers where there will be a juggernaut of connectivity of prices, commodities and customers.

The smart manufacturing sector is improving dynamically. If it is a third-party-based authorization and centralized industrial networks then this is a negative point for this sector because the centralized manufacturing infrastructure leads to lower efficiency, lesser flexibility, lower scalability, and lower security. To get rid of it, obviously, BCoT should be improved as well as sensors.

In this context, we will discuss about the reaction of customers towards sensors in retail markets, introducing a measurement system using quick sort to detect a value within short time and easily. BCoT always had issues with time management as huge data are being sort to detect one. BCoT based sensors also should be more capable to work efficiently within wide range to use in shopping centers. To make it easier we would like to introduce this measurement system which is usable in many cases as it is connected to detect a value within short time.

It can furnish intuitive solutions to address the challenges and issues to reduce limitations of BCoT-based sensors. On the other hand, it will offer customers to attract more as they will feel more comfortable and they will get better efficiency and service. Services always attract customers to any business or institutions. If service can satiate customer, they become regular to get the service and always look for the same service wherever they go. They compare with other services and rely on particular. With this farsighted possibilities and hope, we would like to enlighten the broader sense of the key points and their relationships.

***Smart manufacturing and sensors:***

Sensors were used in Industries and organizations for long time. The first thermostat was introduced in the late 1880s and infrared sensors have been around since the late 1940s. The ability to detect was a need from long ago. Sensors are smaller, slimmer and stretchable, stronger and smarter, more solid, more capable, cheaper.

The invention of the Internet of Things contributed in revolution of sensors. Its functionality and delivery of various kinds of intelligence and data by using different types of sensors in the whole network of connected devices is making sensors more efficient and smarter. It offers autonomous functionalities by combining a set of sensors and a communication network, devices to share information with one another and boost the efficiency of whole system. These resulted sensors to be an essential part in smart manufacturing.

Therefore, a novel paradigm of measurement system will be mentioned here along with experiments which is also reckoned as quick sort, has been regarded as a promising enabler of a wide range of applications and use cases scenarios.

Before that, I would like to give a glimpse on sensors to clear vision below.

***Temperature Sensor -*** measures the quantity of heat energy that allows a physical change in temperature to be detected from a specific source and translates the data for a device or user

***-Uses:*** utilized in A/C control, freezers, and similar environmental control systems, in manufacturing processes, agriculture, and the health industry to maintain the manufacturing process always optimal and to maximize output or production

***-Sub-categories:*** Thermocouples, Resistor temperature detectors (RTD), Thermistors, IC (Semiconductor), and Infrared sensors are all examples of thermocouples.

***Proximity sensors-*** recognizes the presence or absence of any product within range, or the qualities of that product, and converts it into a signal that can be read by customers or a simple electronic device without coming into touch.

***-Uses:*** utilized in the retail business, cars, and parking lots such as malls, stadiums, and airports

***-Sub-categories:*** Inductive, capacitive, photoelectric, and ultrasonic sensors are all types of sensors.

***Pressure sensor-*** detects pressure fluctuations and decreases in pressure, which are translated to an electronic signal. The amount here is determined by the amount of pressure used.

***-Uses:*** in the manufacturing industry, in the upkeep of whole-house water and heating systems.

***Water quality sensors-*** detect the water quality.

***-Uses:*** in manufacturing, in the maintenance of whole-house water and heating systems

***-Examples:*** Chlorine Residual Sensor, Total Organic Carbon Sensor, Turbidity Sensor, Conductivity Sensor, pH Sensor, Oxygen-Reduction Potential Sensor

***Chemical sensors-*** indicate changes in liquid, find out air chemical changes.

***-Uses:*** employed in industrial environmental monitoring and process control, intentionally or unintentionally released toxic chemical detection, explosive and radioactive detection, recycling operations on the Space Station, pharmaceutical companies and laboratories, and so on.

***-Examples:*** Chemical field-effect transistor, Chemiresistor, Electrochemical gas sensor, Fluorescent chloride sensor, Hydrogen sulfide sensor, Non-dispersive infrared sensor, pH glass electrode, Potentiometric sensor, Zinc oxide nanorod sensor etc.

***Gas sensors-*** monitor changes of the air quality, detect the presence of various gases.

***-Uses:*** used for air quality monitoring, detection of toxic or flammable gas, hazardous gas monitoring in coal mines, oil & gas sectors, chemical laboratory research, manufacturing – paints, plastics, rubber, pharmaceutical & petrochemical, etc.

***-Examples:*** Carbon dioxide sensor, Breathalyzer, Carbon monoxide detector, Catalytic bead sensor, Hydrogen sensor, Air pollution sensor, Nitrogen oxide sensor, Oxygen sensor, Ozone monitor, Electrochemical gas sensor, Gas detector, Hygrometer

***Smoke sensor-*** senses smoke (airborne particulates & gases), and it’s level.

***-Uses:*** in manufacturing industry, HVAC, buildings and accommodation infra to detect fire and gas incidences.

***- Examples:*** Optical smoke sensor (Photoelectric), Ionization smoke sensor.

***Infrared sensor-*** sense certain characteristics of its surroundings by either emitting or detecting infrared radiation, measure the heat being emitted by the objects.

***-Uses:*** used in a variety of IoT projects, in Healthcare, smart watches, in smartphones, home appliances, remote control, breath analysis, Infrared vision (i.e. visualize heat leaks in electronics, monitor blood flow, art historians to see under layers of paint), wearable electronics, optical communication, non-contact based temperature measurements, automotive blind-angle detection.

***Level sensor-*** determine the level or amount of fluids, liquids or other substances that flow in an open or closed system is called Level sensor.

***-Uses:*** used in businesses that work with liquid materials e.g. the recycling industry, the juice and alcohol industry rely on these sensors to measure the number of liquid assets in their possession.

Fuel gauging and liquid levels in open or closed containers, sea level monitoring and tsunami warning, water reservoirs, medical equipment, compressors, hydraulic reservoirs, machine tools, beverage and pharmaceutical processing, high or low-level detection, and so on are some of the best use cases for level sensors.

***-Examples:*** Point level sensors, Continuous level Sensor

***Image sensors-*** used to convert optical images into electronic signals for displaying or storing files electronically.

***-Uses:*** used in digital camera & modules, medical imaging and night vision equipment, thermal imaging devices, radar, sonar, media house, Biometric & IRIS devices.

***-Examples***: CCD (charge-coupled device), CMOS (complementary metal-oxide semiconductor) imagers, in the car industry, in IoT industry, in improved security systems, in the retail industry, these sensors serve to collect data about customers, helping businesses get a better insight into who is actually visiting their store, race, gender, age are only some of the useful parameters that retail owners get by using these IoT sensors.

***Motion detector -*** detect the physical movement (motion) in a given area and it transforms motion into an electric signal; motion of any object or motion of human beings, decipher different types of movements, making them useful in some industries where a customer can communicate with the system by waving a hand or by performing a similar action.

***-Uses:*** In the security industry, it is used for intrusion detection systems, automatic door control, boom barriers, smart cameras (i.e. motion based capture/video recording), toll plazas, automatic parking systems, automated sinks/toilet flushers, hand dryers, energy management systems (i.e. Automated Lighting, AC, Fan, Appliances Control), and so on (assistance with making the right purchase decision).

***-Examples:*** Passive Infrared (PIR), Ultrasonic, Microwave.

***Accelerometer-*** to measure the physical or measurable acceleration experienced by an object due to inertial forces and converts the mechanical motion into an electrical output. It is defined as rate of change of velocity with respect to time

***-Uses:*** in cellular and media devices, vibration measurement, automotive control and detection, free fall detection, aircraft and aviation industries, movement detection, sports academy/athlete behavior monitoring, consumer electronics, industrial & construction sites, vibration, tilting, and acceleration detection in general, for monitoring driving fleet, and so on.

***-Examples:*** Hall-effect accelerometers, capacitive accelerometers, piezoelectric accelerometers

***Gyro sensors -*** measure the angular rate or angular velocity is known as Gyro sensors, Angular velocity is simply defined as a measurement of speed of rotation around an axis, in navigating and measuring angular and rotational velocity in 3-axis directions. The most important application is monitoring the orientation of an object.

***-Uses:*** Used for the automation of some production processes in automotive navigation systems, game controllers, cellular & camera devices, consumer electronics, robotics control, drone & RC control helicopter or UAV control, vehicle control/ADAS, and many more.

***-Examples:*** Rotary (classical) gyroscopes, Vibrating Structure Gyroscope, Optical Gyroscopes, MEMS (micro-electro-mechanical systems) Gyroscopes.

***Humidity sensor-*** follow the use of temperature sensors and detect the change in humidity almost instantaneously.

***-Uses:*** Controlling heating, ventilation, and air conditioning systems in the industrial and residential domains Automobiles, museums, industrial spaces, greenhouses, meteorological stations, and so on Paint and coatings industries, hospitals, and pharmaceutical companies to safeguard pharmaceuticals

***Optical sensor-*** measures the physical quantity of light rays and convert it into electrical signal which can be easily readable by user or an electronic instrument/device is called optical sensor.

***-Uses:*** used in healthcare, environmental monitoring, energy, aerospace, and many other industries, oil companies, pharmaceutical companies, and mining companies, in ambient light detection, digital optical switches, optical fiber communications, best suited for oil and gas applications, civil and transportation fields, high speed network systems, elevator door control, assembly line part counters, and safety systems.

***-Examples:*** Photo detector, Fiber Optics, Pyrometer, Proximity & Infrared

***Proximity sensor***

The proximity sensor can sense the proximity of an object (e.g. in an automated production line) which can detect without touching the detection object. The movement information and existence information of the detected object are converted into electrical signals.

***-Uses in details:***

Proximity sensors are used in parking lots in consumer devices (used as capacitive touch switches on consumer electronics products e.g. to detect if a user is holding their phone near their face), in shopping malls, stadiums, and airports to indicate parking spaces, on assembly lines in chemical, food and many other types of industries (e.g. as a diffuse sensor in a public washrooms or as a collision detection sensor for robots).

Proximity Sensors are extensively used in industrial (e.g. for part detection in an industrial conveyor system) and manufacturing applications, exclusively for safety and inventory management applications for object detection, positioning, inspection and counting.

These types of sensors usually emit electromagnetic fields or radiation beams, such as infrared rays. In retail, proximity sensors can detect the movement between the customer and the product he or she is interested in. The user can be notified of any discounts or special prices for products near the sensor. Capacitive proximity sensors, on the other hand, are not limited to metallic targets. These proximity sensors are skilled at detecting anything that can carry an electrical charge.

***Let’s go through some features of proximity to increase lucidity:***

·         Contactless (ensuring object stays well-conditioned, detect both versatile metallic and non-metallic objects, including liquid, powders, and granular)

·         Natural by surface colors of objects (mainly detects physical changes)

·         Usable for humid conditions and wide temperature range usage, contrasting traditional optical detection.

·         Cheaper in price and Service life is longer as compared to other sensors because we do not need to change any parts

·         A higher speed response rate

·         Emit a light beam using high-end photoelectric technology that’s capable of detecting all sorts of objects (photoelectric)

Depending upon some factors, we choose the correct sensors for our purpose and aim. For example, **Object**(Object color, Shape of object, Object material etc.), **Environment** (Cleanliness, Temperature, Moisture etc.), **Range/Distance** (Distance (long or short) between object placed and sensor).

Some of the sub-categorized Proximity Sensors are Inductive Sensors (for metal only, short distance, Suited to be used in harsh environment conditions (to an extent), used in **Industry** Machinery, Automations), Capacitive Sensors (for Metallic and non-metallic objects including liquid, powders, and granular, Short distance,**used in Industry**, Machinery, Automations, Liquid and moisture, Touch sensing, Extremely suitable to be used in harsh environment conditions), Photoelectric Sensors (Object with simple surfaces, Long distance, used in Distance measurement Anemometers for wind speed and direction detection, Automation production processes, Fluid detection, Unmanned aerial vehicles (UAVs) for object monitoring Robotics, Suited for harsh environment conditions except vacuum), and Ultrasonic Sensors (for Object with simple/complicated surfaces, Long distance, used in Item counter, Security systems such as surveillance, burglar alarms, etc.,  
Monitoring and control applications, Not suited to be used in harsh environment conditions) etc.

Among all, we will pick up capacitive proximity where we can implement our measurement application. Here’s description of this sensor given below:

***Capacitive proximity*** sensor produces an electrostatic field when any conductive/non-conductive object (including glass, plastic, water, wood, metals, and a myriad of targets of other materials) reaches the target area, the capacitance of both plates’ increases, resulting in oscillator amplitude gain which generates sensor output switch.

***-Uses:*** **Industrial usages** for example-Production automation machines that count products, product transfers, Filling processes, pipelines, inks, etc., Fluid level, composition, and pressure, Moisture control, Non-invasive content detection, Touch applications and so on.

We already know the characteristics of proximity sensors. Capacitive has the same. But if we discuss the drawbacks of these sensors, then comparatively low range and high prices come up.

There is another type of proximity sensor named magnetic which can detect magnetic objects. Having large ranges, it acquires more qualities e.g. precision, being programmable and cost-effective. A typical one incorporates glass and metal blades that can do quick magnetizing.

***Customers’ reaction towards sensors and smart technologies based on BCoT:***

Before starting the topic, let us understand the definition of the words we need to know.

An IoT device is a device embedded with at least one sensor that uses a wireless communication protocol to connect to and share data from the at least one sensor with other devices directly or indirectly. For example, via an IoT hub or smart device, such as a smartphone.

Sensor can be a component of an IoT device, or a stand-alone object (a radio-frequency identification (RFID) which can be read by an RFID reader) which detects changes. If we think about smartphones, we will see, it uses a number of sensors, such as a timer, accelerometer, and global positioning system (GPS) sensor.

Consumer/ Customer can be indicated as an entity who purchases a product or interacts with a product for the purpose of purchasing the product or does both or equivalents. Normally, a consumer/ Customer makes decisions on the purchase of a product on the basis of the information available online or at a brick-and-mortar store.

Consumer device/ Customer device is owned by a consumer/customer which is capable of communicating with IoT devices enhanced with sensors or without sensors.

Retailers provide products for consumers/ customers for purchasing having or not having a brick-and-mortar presence. He can play a role as a manufacturer of a product or any other person within the distribution chain of the product.

Marketer provides information (coupon or alert) on products to customers. The marketer or the retailer can collect data on consumers and use that data for marketing purposes.

 When customers visit brick-and-mortar stores to purchase anything they may search anyone to assist in their purchasing decisions of the products. Customers examine the product by picking up the product, reading its label, or viewing a review of the product on a website before making a decision to choose and buy (online or offline). IoT devices are located in the supermarkets as well as in the customer devices (such as smartphone, smart glasses, smart watch etc.) which collects data from customer devices for the retailer or marketer. On the other hand, the customer’s smartphone can determine the location of the person near a product based on a proximity sensor on the product to determine the possibility of interacting with the product.

Right after receiving data, the sensor data is analyzed by the retailer or marketer to determine the interactions between the consumer and an IoT device. To map some interactions, it may require specific sensor data to determine that the consumer has performed that particular communication. The received sensor data is examined to check whether it matches with the specific sensor data mapped to a specific interaction or not. If it is matched then we understand that the interaction is identified.

Customer devices also help to know the information about the interaction between customer and the sales associate. If the signals from the product and sales associate indicate that the consumer is in close proximity to the product and sales associate for a predetermined time, it may be determined that the consumer is interacting with the sales associate about the product.

In customer navigation tech (e.g. **Bluetooth Low Energy (BLE)**) or customer devices, machine-learned models use historical information regarding interaction. Each customer device specifies specific data. For any specific customer, input, output lead score is counted.

A standard conversion rate can be calculated on the basis of any kind of interaction between product and customer that customers bought any particular product after examining, out of the other customers who have come across the details.

Finding customers’ choices, finding best prices for the product by customer, costing determination by retailer for example garments manager or financial advisor, finding average cost or average values of complicated rates or functions, calculating the most appropriate and relevant scores or costs or values after updating the data or getting additional new data with any specific interval of time, finding the probability of the best choice i.e. proximity of the consumer to the product on the basis of previous calculations for predetermination of cost or choice can be performed by any invention described in the form of computer code or machine-useable instructions, including computer-executable instructions such as program components, being executed by a computer or other machine, such as Bluetooth Low Energy (BLE),a personal data assistant or other handheld device. We know that program components, including routines, programs, objects, components, data structures or code performs particular tasks or implements particular data types. Performance expectancy, Effect expectancy, Social influence, Facilitating conditions, motivation, Price Value, Habit etc. factors influence customer to install a retailer’s application on their IoT devices enhanced with sensor, to use the mobile application in a retail store, to allow their location to be used for services rendered to them and increase demand, facility towards customer.

We can implement the proposed calculation ***(i.e. using quicksort in sorting precise memory in sensor)*** in existing different aspects of the inventions which may be practiced in a variety of system configurations, including handheld devices, customer devices, BcoT- based electronics, computers, computing devices, etc.

For example, Qualcomm released proximity beacons and Apple released their trademarked iBeacon (broadcasts 3 values: unique ID, major ID and minor ID). Both these products are compatible with the BLE technology stack and be used for indoor location tracking.

The program can be used in distributed computing environments. The advantage of this will be performing any work remotely linked through any communications network. Customers will be benefited by using the system as well as retailers or marketer will get smoother sensors or devices.

The transaction time for data collection and sorting is one of the main factors to improve performance of some real-time BIoT applications. However, computational and communication capacities of processing computers give lower limits to transaction time. To break this limitation, we propose an efficient sorting huge amount of data method using a progressive quality improvement approach.

Only the cases where higher quantity and quality data are needed for analyses, the processing BIoT devices collects and sort them. Our proposed method reduces the average transaction time as it is more efficient program.

The use of IIoT networks in the industrial domain such as smart manufacturing factories, ICS, SCADA can cause a disaster or a loss of money if they do not initiate their function at the proper time. Which shows, we need to use such a program which takes less time within the performance as well as updating blocks. Smoother function or coding is always helpful for any digital devices.

*Idea about sorting, uses and advantages of using it in the memories of BCoT and sensors:*

To expand traditional approximate computing to a broader scope, we can leverage the approximate memory to improve the performance of sorting algorithms, but still producing precise results.

Particularly, we know three classic and popular sorting algorithms: quicksort, mergesort, and radixsort (the first two are comparison-based, while the last is not) using approximate main memory. Simulation results show that without the requirement of precise outputs, quicksort and radixsort can get a nearly sorted sequence while reducing 30% to 40% write latencies on the approximate memory.

Apart from focusing on approximate computing with approximate hardware, we wrote a fast sorting algorithm on the memory system with both precise and approximate memory for guaranteeing precise results. We use a novel algorithmic level execution mechanism on hybrid approximate/precise memory to produce precise results. Specifically, we propose program mechanism in which the approximate memory acts as an accelerator. If anyone copy the input data from the precise memory to the approximate memory, and then perform an existing sorting algorithm on the approximate memory even then the approximate results is possible to be precised in the precise memory. If the sorting algorithm can deliver a nearly sorted output on approximate memory, only a lightweight IoT device is needed afterwards. As a result, the cost of devices and data copies between precise memory and approximate memory can be compensated by the gain of uploading the sorting algorithm to the approximate hardware. Approximate hardware can also be used for improving the performance of precise computing, which broadens the application scope of approximate hardware.

Once upon a time, approximate hardware was only used for approximate computing. **As a result, we develop and test commonly used sorting algorithms on hybrid storage systems with precise and approximate storage, as well as show system and architectural insights for enabling precise computation on approximate hardware.**

Though system interfaces should be carefully redesigned to support hybrid approximate/precise main memory, the modification in hardware is lightweight and easy to implement.

***This algorithm is well suited to database and data mining applications. To implement an efficient bitonic sorting network, we use the texture mapping and blending capabilities of GPUs.***

***Meanwhle, in order to improve overall sorting performance, we describe an efficient instruction dispatch mechanism and an efficient memory data access pattern in our new algorithm. Our sorting algorithm has been used to speed up join-based queries and stream mining algorithms.***

***The results show an order of magnitude improvement over previous CPU and GPU-based sorting algorithms.***

Sorting is a well studied problem in the theory of algorithms. In fact, optimized implementations of some algorithms such as Quicksort are widely available. These include optimized implementations available as part of standard compilers such as Intel C++ compiler and Microsoft Visual C++ 6.0 compiler. The implementation of Quicksort in the Intel compiler has been optimized using Hyper- Threaded technology. More details on the implementation of Quicksort are given here1. In the database literature, many fast algorithms have also been designed for transaction processing and disk to disk sorting [1]. However, the performance of sorting algorithms on conventional CPUs is governed by cache misses and instruction dependencies.

In terms of using GPUs for sorting, Purcell et al. described an implementation of bitonic merge sort on the GPUs. The bitonic sort is implemented as a fragment program and each stage of the sorting algorithm is performed as one rendering pass. Kipfer et al. [24] presented an improved bitonic sort routine that achieves a performance gain by minimizing the number of instructions in a fragment program and the number of texture operations, but the algorithm still requires a number of instructions. More recently, Govindaraju et al. [19] presented an improved sorting algorithm using texture mapping and blending operations, which outperforms earlier GPU-based algorithms

The average case time complexity of quicksort is **O(n\*logn)**.

**Quick sort** is the better suited for large data sets. It is the fastest and efficient algorithm for large sets of data. But it is inefficient if the elements in the list are already sorted which results in the worst case time complexity of O(n2).

**Quicksort** is one of the most efficient sorting algorithms, and this makes of it one of the most used as well. The first thing to do is to select a pivot number, this number will separate the data, on its left are the numbers smaller than it and the greater numbers on the right

Which sorting algorithm takes more memory space?

Quick Sort and **Insertion Sort** both are in-place and comparison-based sorting algorithms. None of the above two sorting algorithms demand extra space from us.

No other sorting algorithm performs better than Quick sort on Arrays because of the following reasons: **Quick sort is an in-place sorting algorithm, i.e. which means it does not require any additional space**, whereas Merge sort does, which can be rather costly.

since it has the upper hand in the average cases for most inputs, **Quicksort** is generally considered the “fastest” sorting algorithm.

**How to Boost QuickSort Performance?**

1. Better pivot selection. ...
2. Hoare's Partitioning Scheme. ...
3. Handle Repeated elements. ...
4. Using Tail Recursion. ...
5. Hybrid with Insertion Sort.

Which is better randomized quick sort or quick sort?

The advantage of **randomized quicksort** is that there's no one input that will always cause it to run in time Θ(n log n) and the runtime is expected to be O(n log n)

Which sort is best for array?

Quicksort. Quicksort is generally thought of as the most efficient 'general' sorting algorithm, where nothing is known about the inputs to the array, and it's more efficient than **insertion sort** on large lists.

Does quicksort need external memory?

Abstract. An external sorting algorithm based on quicksort is presented. The file to be sorted is kept on a disk and **only those blocks are fetched into the main memory which are currently needed**.

The sorting algorithm is used for information searching and as Quicksort is the fastest algorithm so it is widely used as a better way of searching. It is used **everywhere where a stable sort is not needed**. Quicksort is a cache-friendly algorithm as it has a good locality of reference when used for arrays

How fast is quicksort?

The time complexity of Quicksort is O(n log n) in the best case, O(n log n) in the average case, and O(n^2) in the worst case. But because it has the best performance in the average case for most inputs, **Quicksort is generally considered the “fastest” sorting algorithm**.

A good reason why Quicksort is so fast in practice compared to most other O(nlogn) algorithms such as Heapsort, is because **it is relatively cache-efficient**. Its running time is actually O(nBlog(nB)), where B is the block size

How does quick sort work? Quicksort is a fast sorting algorithm that works by **splitting a large array of data into smaller sub-arrays**. This implies that each iteration works by splitting the input into two components, sorting them, and then recombining them.

Does quicksort work with duplicates? When the list of items to be sorted contains a lot of duplicate values, **we can improve QuickSort by grouping all the values that are equal to the pivot to the middle and then we recursively QuickSort those values on the left and those values on the right**.

Quicksort is one of the most efficient sorting algorithms, and this makes of it one of the most used as well. The first thing to do is to select a pivot number, this number will separate the data, on its left are the numbers smaller than it and the greater numbers on the right. With this, we got the whole sequence partitioned. After the data is partitioned, we can assure that the partitions are oriented, we know that we have bigger values on the right and smaller values on the left. The quicksort uses this divide and conquer algorithm with recursion. So, now that we have the data divided we use recursion to call the same method and pass the left half of the data, and after the right half to keep separating and ordinating the data. At the end of the execution, we will have the data all sorted.

Main characteristics:

* From the family of Exchange Sort Algorithms
* Divide and conquer paradigm
* Worst case complexity O(n²)

**Why is Quick Sort preferred for Arrays?**

No other sorting algorithm performs better than Quick sort on Arrays because of the following reasons:

* Quick sort is an in-place sorting algorithm, i.e. which means it does not require any additional space, whereas Merge sort does, which can be rather costly. In merge sort, the allocation and deallocation of the excess space increase the execution time of the algorithm.
* The locality of reference is one of the critical reasons for quick sort’s efficiency. It allows the computer system to access memory locations close to each other, which is faster than memory locations distributed throughout the memory, as in the case of merge sort.
* Quick sort is most commonly implemented using a randomized version with anticipated time complexity of O(NlogN). Although the worst case is possible in the randomized version, it does not occur for a specific pattern (such as a sorted array). Hence the randomized quick sort works well in practice.

   
The quick sort is faster than merge sort in many cases because of reduced overhead but because of how quicksort accesses data, which is a lot more cache-friendly than standard mergesort.

It is used on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations as it is not difficult to implement. It is a good general purpose sort and it consumes relatively fewer resources during execution.

## Advantages

* It is in-place since it uses only a small auxiliary stack.
* It requires only ***n (log n)*** time to sort **n** items.
* It has an extremely short inner loop.
* This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

The time complexity of Quicksort is **O(n log n)** in the best case, **O(n log n)** in the average case, and **O(n^2)** in the worst case. But because it has the best performance in the average case for most inputs, Quicksort is generally considered the “fastest” sorting algorithm. At the end of the day though, whatever the best sorting algorithm really is depends on the input (and who you ask).

Sorting is the process of organizing elements in a structured manner. Quicksort is one of the most popular sorting algorithms that uses nlogn comparisons to sort an array of n elements in a typical situation. Quicksort is based on the **divide-and-conquer strategy.** We’ll take a look at the Quicksort algorithm in this tutorial and see how it works.

**The Idea of QuickSort**

Quicksort is a fast sorting algorithm that works by splitting a large array of data into smaller sub-arrays. **This implies that each iteration works by splitting the input into two components, sorting them, and then recombining them.** For big datasets, the technique is highly efficient since its average and best-case complexity is O(n\*logn). It was created by Tony Hoare in 1961 and remains one of the most effective general-purpose sorting algorithms available today. It works by recursively sorting the sub-lists to either side of a given pivot and dynamically shifting elements inside the list around that pivot. can be summarized in three steps **Pick:**Select an element.**Divide:** Split the problem set, move smaller parts to the left of the pivot and larger items to the right. **Repeat and combine:** Repeat the steps and combine the arrays that have previously been sorted.

**Benefits of Quicksort**

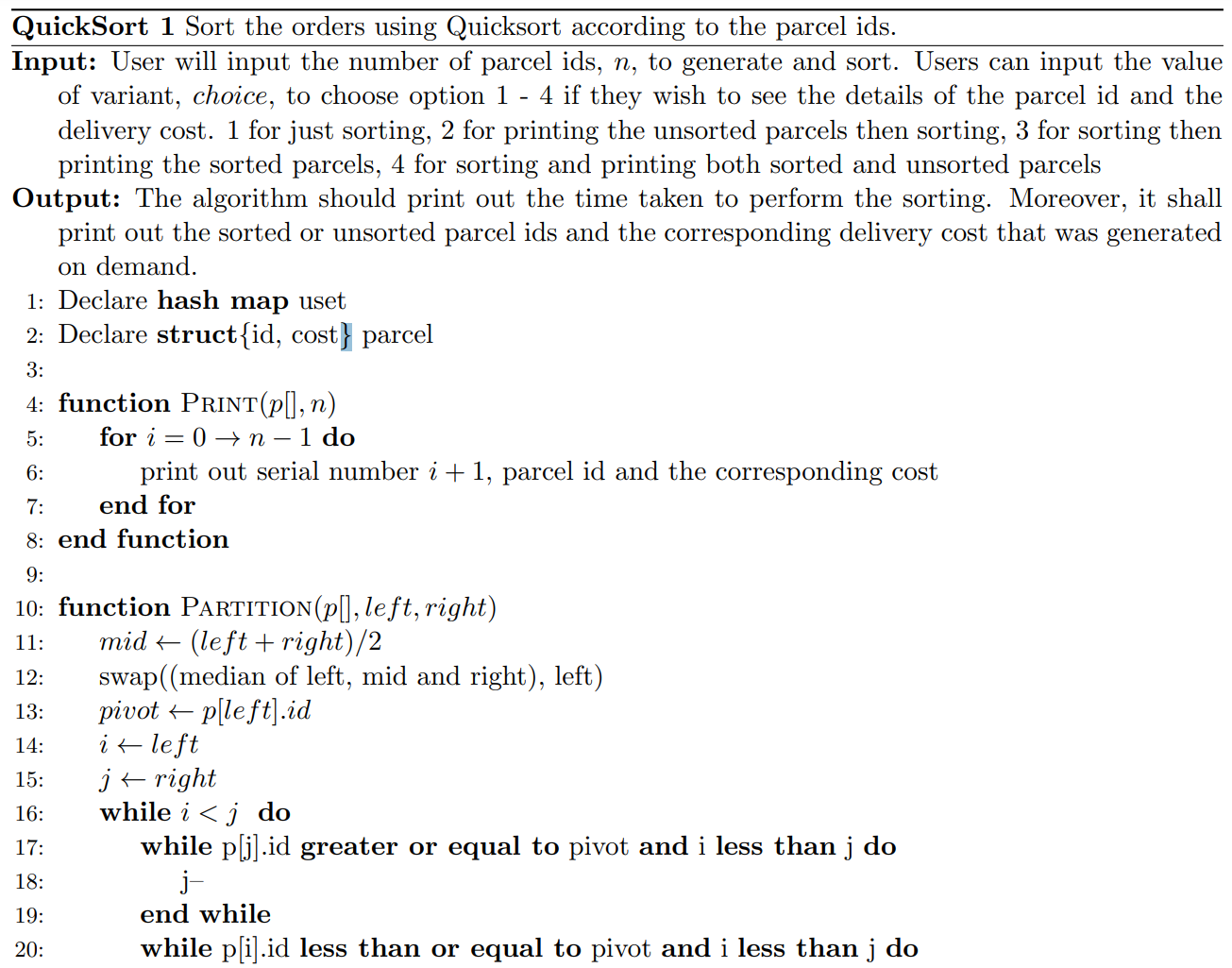
Let’s go through a few key benefits of using Quicksort: It works rapidly and effectively. It has the best time complexity when compared to other sorting algorithms. Quick sort has a space complexity of O(logn), making it an excellent choice for situations when space is limited.

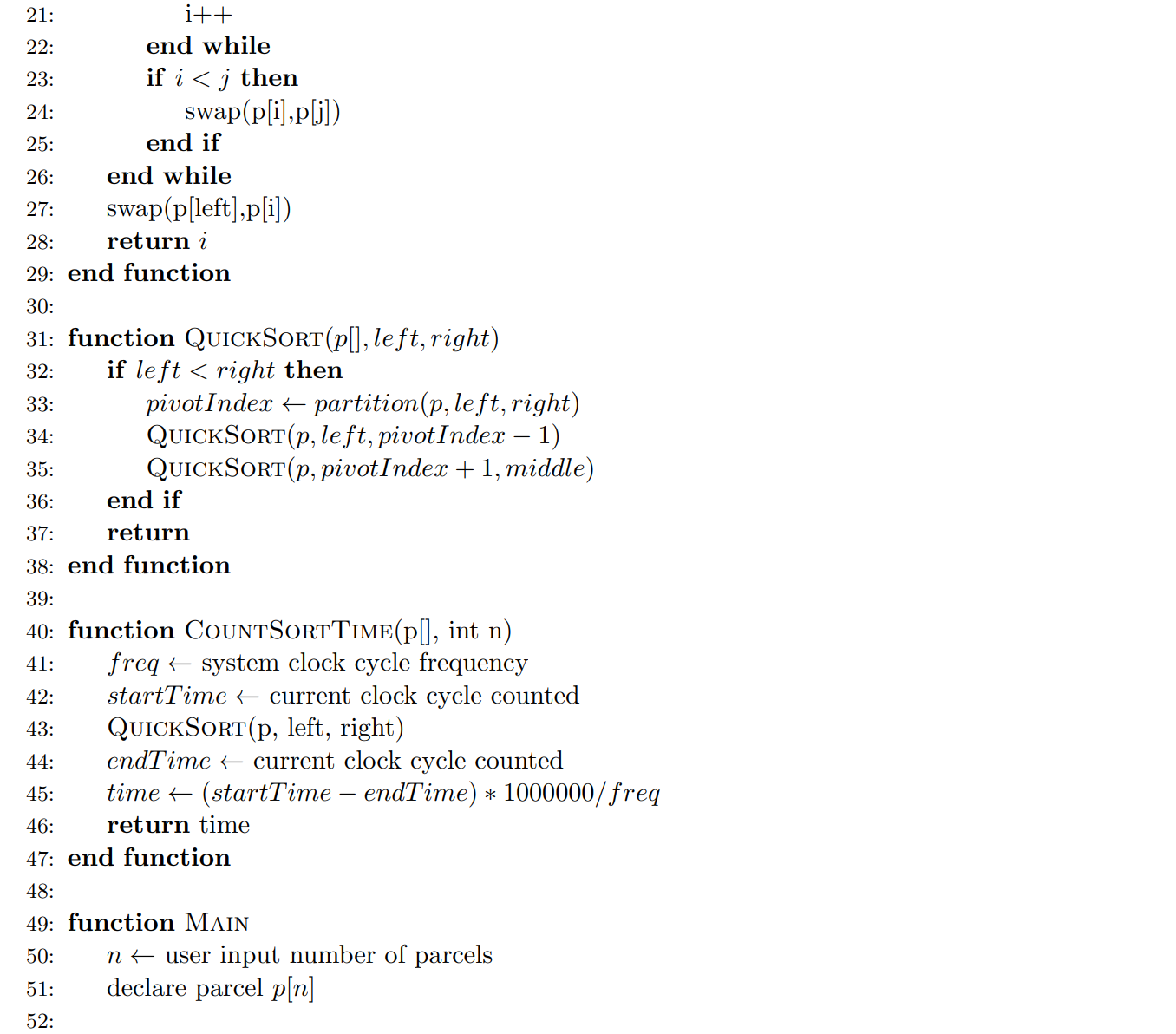
***Using quicksort in sorting precise memory in sensor***

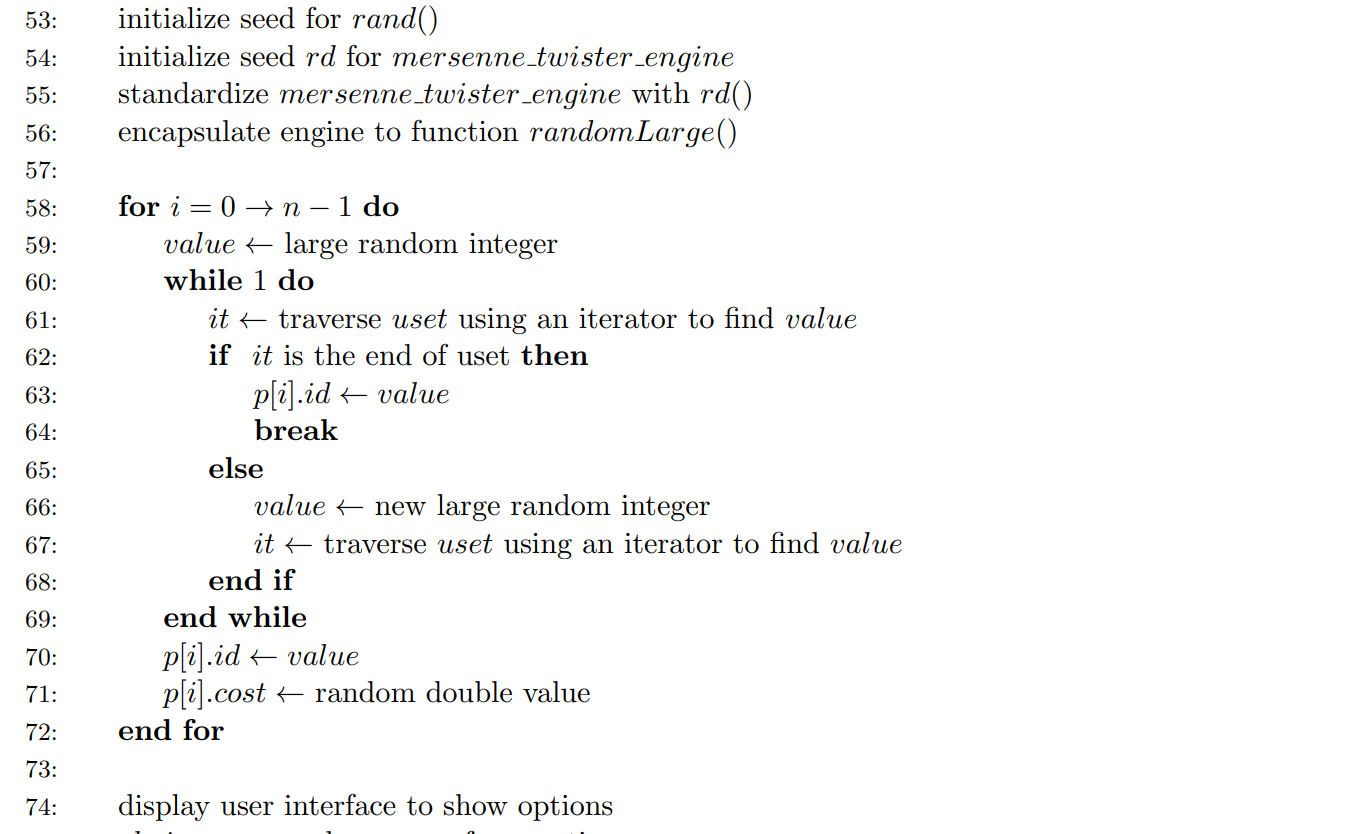
# *Quicksort: Theory and Experiments*

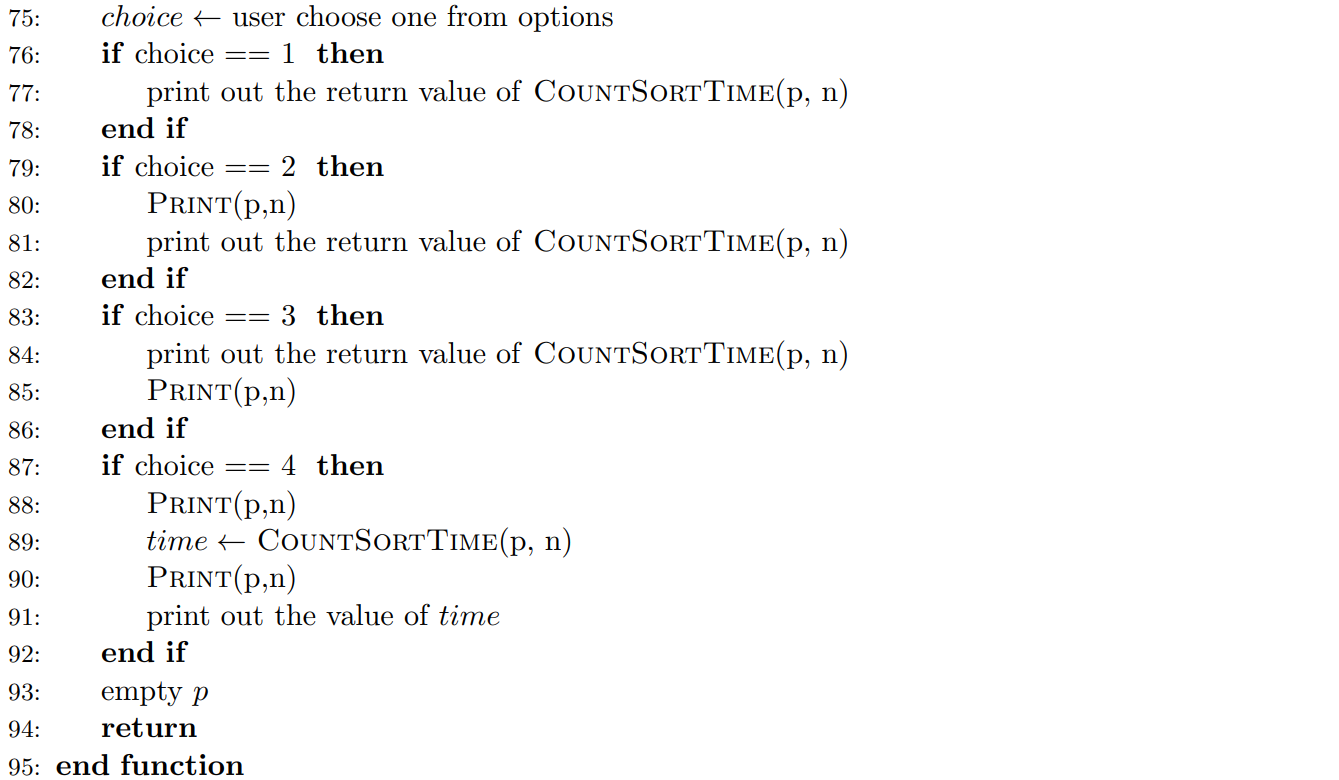
## *Pseudocode*

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 pseudo code.









## 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).

**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;  
}

***Future challenges and possibilities:***

General Beacon devices or **Bluetooth Low Energy (BLE) [13]** or any BCoT- based sensors uses improvable programs which can be more effective after improving. Retailers, marketers or industrialists can use data or can perform any operating using improved programs easily.

Limitations of the customer devices or Block chain or IoT or sensors are available from the circumstance of software and hardware. Being a hardware, IoT, Customer devices, Sensors sometimes are not capable enough to let a better program or block chain to be installed. People started thinking about lightweight block chain to make it compatible with hardware devices they are needed to be installed. In the same way improved software can be installed into the devices which will make devices more efficient and there will be no need of replacing devices or generating new devices if software or block chain is modified to make compatible for the existing devices. It will resulted into no wastage of devices which may not do cope-up with the advanced software or block chain or could not keep pace with the dynamic advancement of digital world.

Future challenges of make balance between devices i.e. and soft wares need more professional attachments to the research works going on widely. It can save huge amount of wastage of devices, time and will make devices, services more convenient and more secured to people. Thus economic growth in buying and selling will be increased.

Obviously without these customer devices, Block chain, IoT and sensors, we cannot think about dynamic advancement in industrial, agricultural or in any other sectors or stormy revolution in technological world now a days. When all are combined in smart manufacturing to manufacture incredible inventions, software is also needed to compete with the journey of the combinations of all of the hardware.

As for future work, the approx-re\_ne scheme has raisedan interesting design strategy for leveraging approximate hardware for database systems. This paper makes the \_rst and important step of utilizing approximate hardware for MSD. database operations. More research e\_orts have to be made in the future, e.g., other database operations (such as aggregations) on approximate hardware and reducing the overhead of re\_nement stage.

We have presented a cache-efficient sorting algorithm that maps well to the GPUs. We analyzed the memorylimitations of current GPU-based sorting algorithms and presented techniques to improve the computational performance.

Furthermore, our new sorting algorithm performs fewer number of memory accesses and exhibits better locality in data access patterns. It takes into account the sorted nature of the input sequence and improves the overall performance.

We have demonstrated its application on some database and data mining applications. Our results indicate a significant improvement over prior techniques.

There are many avenues for future work. We would like to investigate new features and capabilities of GPUs that can further improve the performance, and compare against other optimized algorithms such as hash-joins on CPUs. We would like to use our sorting algorithm for other data mining applications. Furthermore, we would like to develop cachefriendly GPU-based algorithms for other computations including nearest neighbors and support vector machine computations.

***Conclusion:***

Advantage, disadvantages all will walk side by side while we see advantages of all inventions mentioned above are huge comparatively disadvantages. Disadvantages or limitations come up whenever we try to improve a system or device keeping pace with the dynamic world in accordance with our need. Improving infrastructure, introducing novelty we can ensure enhancement in the systems we use. This paper describes our little effort in giving idea of the highly influential components of digital world i.e. customer devices, Block chain, IoT, sensors, smart manufacturing etc. and factors of financial field, measurement calculations and possible improved program, relationships between them. Obviously there are more areas to improve. In future, we hope we will be able to come up with something more to mitigate limitations.

***References***

1.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

1. 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
2. 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
3. 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
4. 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
5. 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
6. 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
7. 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
8. Shaffer, C. (2012). *Data Structures and Algorithm Analysis in C++*. Dover Publications.
9. 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
10. 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
11. 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
12. https://www.researchgate.net/publication/316899615\_Consumer\_attitudes\_towards\_proximity\_sensors\_in\_the\_South\_African\_retail\_mark