UNIVERSITY OF VAASA

FACULTY OF TECHNOLOGY

TELECOMMUNICATION ENGINEERING

Yuan Yue

Performance Evaluation of Sorting Algorithms in Raspberry Pi and Personal Computer

Master´s thesis for the degree of Master of Science in Technology submitted for inspection, Vaasa, 6 August, 2015.

Supervisor Prof. Mohammed Elmusrati

Instructor Tobias Glocker

ACKNOWLEDGEMENT

This thesis is aimed to study the different time complexity of four sorting algorithms in both Raspberry Pi and personal computer.

First of all, I would like to express my sincere gratitude to my thesis instructor Tobias Glocker for his constant guidance and patient instruction during my thesis research. Moreover, I should present my great appreciations to Professor Mohammed Elmusrati and the staffs in University of Vaasa who has provided me the essential devices for my thesis. At last, I would like to express great thanks to my families and friends who give me encouragement and support.

TABLE OF CONTENTS **PAGE**

ABBREVIATIONS

CPU	Central processing Unit
GPIO	General-purpose input/output
GUI	Graphical User Interface
IDE	Integrated Development Environment
IP	Internet Protocol
IT	Information Technology
MB	Megabyte
NTP	Network Time Protocol
PC	Personal Computer
PHP	Hypertext Preprocessor
PRNG	Pseudorandom Number Generator
RAM	Random-Access Memory
SDRAM	Synchronous Dynamic Random-access Memory
SSH	Secure Shell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
USB	Universal Serial Bus

UNIVERSITY OF VAASA
Faculty of Technology
Author: Yue Yuan
Topic of the Thesis: Performance Evaluation of Sorting Algorithms in Raspberry Pi and Personal Computer
Name of the Supervisor: Professor Mohammed Salem Elmusrati
Instructor: Tobias Glocker
Degree: Master of Science in Technology
Department: Department of Computer Science
Degree Program: Degree Program in Information Technology
Major of Subject: Telecommunication Engineering
Year of Entering the University: 2011
Year of Completing the Thesis: 2015 Page: 78

ABSTRACT

Nowadays, more and more data is going to be collected. Most of the collected data must be sorted to be analyzed. Thus it is important to use efficient algorithms to save time and energy. This is especially important for battery-operated devices that collect and sort data directly. The lifetime of the battery can be remarkably extended by choosing effective sorting algorithms. Furthermore, processors with lower clock frequencies can be used, leading to lower battery consumption.

This thesis focuses on the time complexity of algorithms (bubble sort, insertion sort, quick sort and heap sort) executed on raspberry Pi and personal computer. In this thesis, a client and a server have been implemented. The client is a Graphical User Interface (GUI) that generates random numbers, which are sent to the server. On the server the random numbers are sorted and the sorting time is measured. After sorting, the random numbers will be sent back to the client where they are displayed on the GUI. The sorting time is also send to the GUI in order to display it.

It is to mention that the time consumption is measured with timing functions of the C library. An oscilloscope has been used to check if the timing function works properly. At the end, the performances of four sorting algorithms on Raspberry Pi and on the Personal Computer (PC) are compared.

KEYWORDS: Sorting Algorithm, Socket Communication, GUI, random numbers, efficiency

1. INTRODUCTION

Nowadays, sorting algorithms are regarded as one of the most important areas in computer science because there are more and more data going to be sorted and analyzed. Sorting a large amount of data utilizes substantial amount of calculation resources. There are many kinds of algorithms that have been upgraded and developed. An excellent sorting algorithm can save lots of resources, but to choose a suitable algorithm for different applied environment and requirement is a demanding work. The limits and rules of different data must be considered. For example, the calculation speed of a search engine is related to the number of compared key words. Choosing an efficient sorting algorithm not only can save lots of time but also could cost lower battery energy in devices.

On the other hand, embedded technique is worldwide utilized in a wide variety of technical fields since it is able to integrate into different physical processors. It is different from a general computer system such as personal computers. Embedded system usually executes the programs that have been coded before the equipment is being used. At present, electronic devices are developing rapidly. It is easy to find out that there are many electronic devices with embedded systems in human being's daily life, like electronic watch, digital camera, car etc. Since human life pace is accelerating, working speed and battery lifetime are the keys of electronic devices. As a result, efficiency turns out to play a more and more important role in the Information Technology (IT) area. The efficiency can be improved by choosing an appropriate sorting algorithm. Thus it is necessary to figure out the time complexity of sorting algorithms.

This thesis mainly focuses on discovering the differences between four popular sorting algorithms in different amount of random numbers and comparing the result from PC and Raspberry Pi.

The thesis consists of five chapters. In the first two chapters, the sorting algorithms, embedded system, the background information of the generation of random numbers and boards, which are utilized in the experiment, are introduced. In the third chapter, the theories of sorting algorithms (bubble sort, heap sort, insertion sort and quick sort) are described. Chapter four explains the method of the experiment including the software and hardware implementation, the algorithm execution and the results. The last chapter contains the conclusion and future work.

2. BACKGROUND INFORMATION

2.1 Random Number Generation

It is common to use random number generation in techniques or simulations nowadays, for example, bank account numbers and cryptography. It is a necessary and basic technique in computer programming. Random number is a numeric sequence which contains elements without recognizable and regularities. They are divided into “true” randomness and pseudo randomness. The “true” random numbers are used in cryptography or algorithms require high degree of randomness while pseudo random numbers are satisfied with many other operations that only need unpredictability.

Pseudorandom Number Generator (PRNG) contains base generators and distributional generators. A base generator creates a sequence of values distributed uniformly over the range (0,1), while a distributional generator produce random numbers from a base generator then transform them into different variables according to a specific distribution (Wang, Shen & Zhang 2006; Haahr). Pseudorandom and uniformly distributed numbers are used in the experimental part.

2.2 Socket Communication

A socket enables communications between client and server, which are considered as the endpoints in the communication system. A client socket is installed in a computer and uses Internet Protocol (IP) address of this computer to call a server, which is on the other computer or device. Once the connection is created, client and the server can transmit data to each other.

2.2.1 TCP

Transmission Control Protocol (TCP) is a basic communication language of Internet. It is a reliable, ordered and error-checked transmission with streams of data over the IP network.

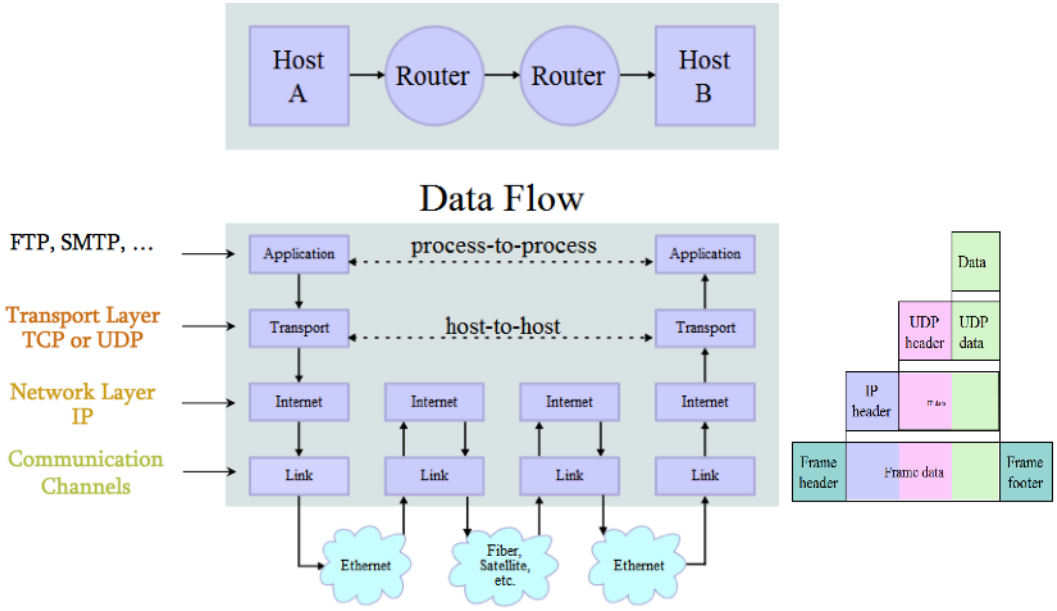

Figure 1. Network topology and layers (Wikipedia Internet Protocol Suite 2015).

TCP is widely used in remote manipulation, file transmission etc. which are depending on the reliable transmission protocol. As the Figure 1 shows, TCP is working in the transportation layer of the network. This protocol guarantees that all the sending bytes would be received in the correct order. It requires the receiver to give feedback with an acknowledgement message when the receiver end receives the stream. The sender also needs to record every sending package and contains a timer to check if the sending message has been acknowledged or not. (Sheldon 2001)

2.2.2 Sockets

At the beginning the socket was designed for the Unix operation system. The data transmission is a special I/O in socket communication. There is a function for opening a socket that is similar to the function open a file. This function will return an integer data type. After opening the socket, other operations such as creating connection, data transmission etc. will be established in the process. Generally, there are two kinds of sockets, datagrams socket and stream socket. Datagram socket uses User Datagram Protocol (UDP) to provide individually addressed and route data transmission without order. On the other hand, stream socket uses TCP to provide an oriented and sequenced connection. Figure 2 shows how sockets work in the network. The data transmission in TCP socket is reliable.

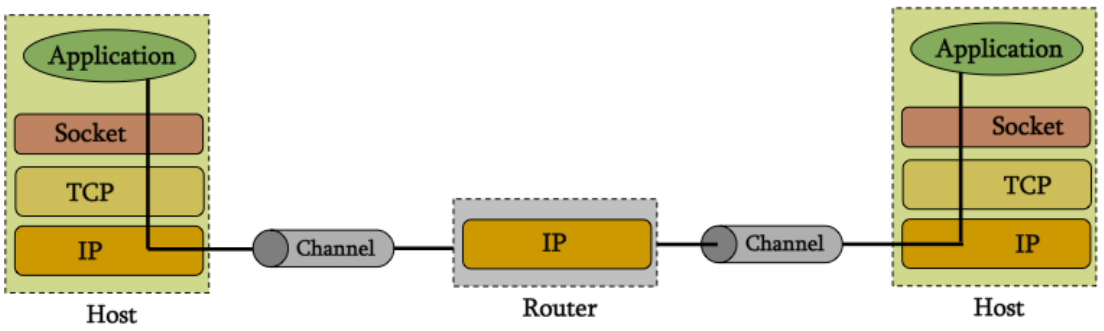

Figure 2. Socket connection through TCP/IP (Fatourou & Kosmas 2012).

Normally, a socket contains executed code on both communication ends. The client side is responsible to start data transmission, while the server is waiting for the connection from remote clients. On the server side, the socket runs on a computer and binds to a specific port number. It waits and listens to the requests from the client. The client must know the hostname or IP address of the server as well as the port number on which the server is listening. The client and server can read and write the

data from the socket after the connection is accepted. Figure 3 shows the procedures of stream sockets communication.

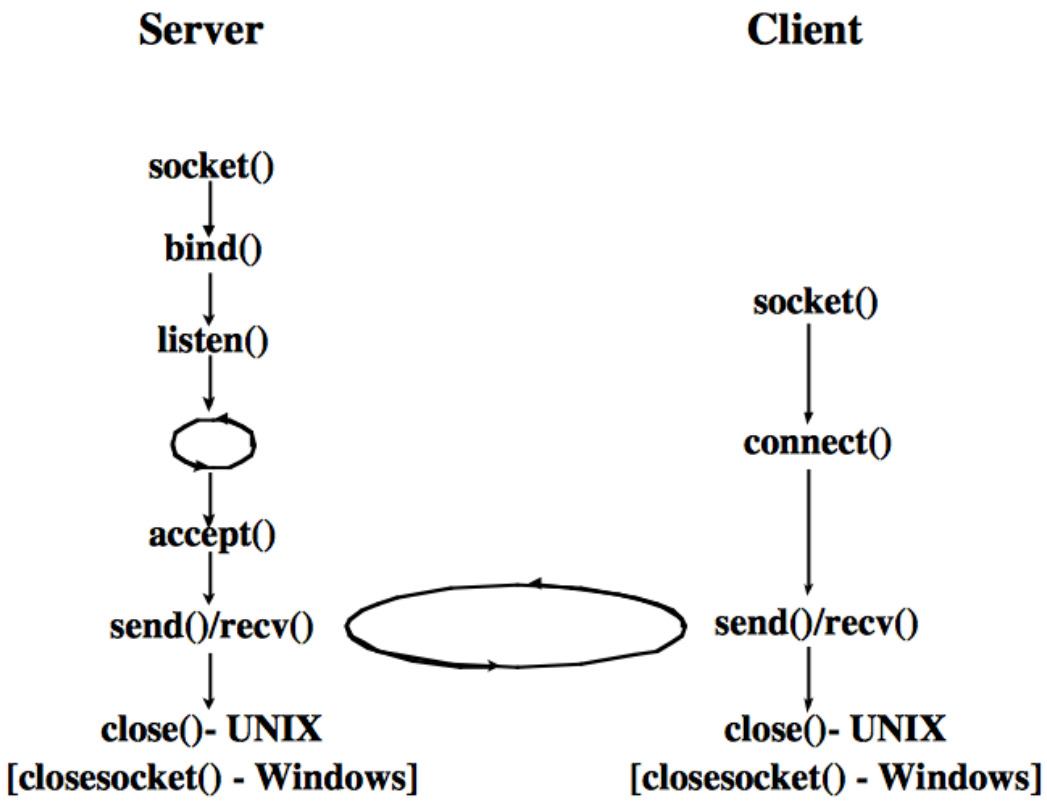

Figure 3. Flowchart of stream sockets communication (Stallings 2007: C–38).

Procedures and functions of sockets communication are discussed below (in C).

- Socket() initiates a new socket of a certain socket type for communication and returns a descriptor.

```
#include <sys/types.h>

#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

The domain argument specifies a communication domain, which includes AF_UNIX, AF_INET, AF_OSI, etc. AF_INET refers to address family and communicates to IP address on the Internet.

The type argument determines the communication semantics. There are three types to choose: SOCK_STREAM, SOCK_DGRAM and SOCK_RAW. SOCK_STREAM is used for TCP reliable oriented connection, while SOCK_DGRAM is used for UDP unreliable datagram. SOCK_RAW is for IP level (Stallings 2007). In the experiment, SOCK_STREAM is utilized because TCP stream socket can guarantee all data would be received safely.

The protocol argument decides a special protocol to use in this socket, but normally there is only one protocol would be use in the socket. In this case, protocol should be specified as 0.

It will return an integer socket descriptor when it is working successfully, otherwise return -1 as a failure. An example is shown as following.

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);

if (sockfd < 0)

{

perror("ERROR opening socket");

exit(1);

}
```

- Bind() is used to bind a name to a socket.

```
#include <sys/types.h>

#include <sys/socket.h>
```

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

When a socket has been created, there will be an address space without address detail. Bind specifies this address by addr to the socket by the file descriptor sockfd. Addr is a pointer to the local address structure of this socket. Addrlen is the length of the structure referenced by addr.

If successful, bind() return 0, otherwise will return -1 on error. Normally, it should be coded as following.

```
if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)

{

perror("ERROR on binding");

exit(1);

}
```

- Connect function establishes a connection on a socket client side. It is required in the stream socket (TCP/IP) which is connection oriented.

```
#include <sys/types.h>

#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

The sockfd is a local socket file descriptor. The addr is a pointer to protocol of socket and the addrlen is the length of the address structure. If connect() is on success, it will return 0. Otherwise it will return -1 as failure.

- The listen function is used to prepare a socket to accept messages from client side in the oriented connection communication. It marks a socket, which is going to accept incoming connection request.

```
#include <sys/types.h>

#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

The backlog argument determines the maximum number of incoming connection requests can be wait to be processed by the server. Listen() will return 0 on success and -1 on failure. For example:

```
if(listen(sockfd,5)<0);

{

perror("ERROR on binding");

exit(1);

}
```

- Accept function is to accept an oriented-connection request from the client side. After the server accepts the connection, the socket is no longer in the listening status.

```
#include <sys/types.h>

#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

The sockfd is the socket file descriptor that has been created with socket() and bound to a local address with bind() and listen to the connection request by listen(). The addr argument is a pointer to a sockaddr structure. The structure includes the address of the

peer socket, which is known to the communication layer. The addrlen argument initializes the size of the structure pointed to by addr.

Accept() blocks the caller until a connection is incoming. If the socket is non-blocking and no pending connections are present on the queue, this function will give back an error EAGAIN or EWOULDBLOCK as failure.

If successful, the system will return a nonnegative integer as the file descriptor. Otherwise it will give -1 feedback as failure. Here is an example.

```
newsockfd = accept(sockfd, (struct sockaddr *)&cli_addr, &clilen);

printf("Connect successful!\n");

if (newsockfd < 0)

{

perror("ERROR on accept");

exit(1);

}
```

- Read() and write() are to transmit the messages to another socket after the client side and the server side have established the connection. In fact, there are more than one pair functions to transmit data through sockets. But using read() and write() is a typical way to utilize without flag argument.

```
#include <unistd.h>

ssize_t read(int fildes, void *buf, size_t nbyte);

ssize_t write(int fildes, void *buf, size_t nbyte);
```

The read function is to read nbyte bytes from the associated file with the open file descriptor fildes and put this nbytes data into the buffer pointed to by buf. There are three different modes when it reads from a stream file: byte-stream mode, message-nondiscard mode and message-discard mode. The default setting is byte-stream mode, in this situation, maximum n bytes of data from the stream. Read function will return the number of bytes that has been read and place the zero-byte message back on the stream and wait for the next read(). The return number is determined by the bytes of data that has been actually read. (IEEE Std 1003.1. 2004.)

```
char buffer_in[1000];

char buffer[100000];

int n;

bzero(buffer, 100000);

while(1)

{

bzero(buffer_in,1000);

n = read(newsockfd,buffer_in,sizeof(buffer_in));

if (n < 0)

{

perror("ERROR reading from socket");

return 1;

}

strcat(buffer, buffer_in);
```

```
if(buffer_in[n-2] == '\0')

break;

}

printf("received data is %s.\n", buffer);
```

The write function writes a certain number (nbytes) of bytes from the buffer pointed by buf to the file associated with fildes. If nbyte is smaller than the size of a write buffer, write() will only send out nbyte bytes data out and return nbyte. Practically, if the value of nbyte is greater than the write buffer size of the TCP/IP socket, write() will send as much data as possible and return the number of bytes has been sent. Then this program must issue more calls of write for the remaining data. Usually, it is necessary to add an end byte in the message in order to make read function to check if this is the end of the whole message, then read function would stop and execute the following programs.

For example, there are 100000 bytes that should be sent. The code would be written as following.

```
char buffer[100000];

char *ptr = (char *)buffer;

while (lengthOfString > 0)

{

int n = write(newsockfd,ptr,lengthOfString);

if (n < 0)

perror("ERROR writing to socket");
```

```
ptr += n;

lengthOfString -= n;

}
```

- Close() is used to terminate a socket connection and free the resources allocated to that socket. (IEEE Std 1003.1. 2004)

```
#include    <sys/socket.h>

int close(int fildes);
```

If the socket could be closed, close() returns 0. Otherwise it returns -1 to indicate an error. The close() would fail if the fildes argument is not valid, or it was interrupted by a signal, or the socket is still reading or sending a message.

2.3 Raspberry Pi Board

In this thesis, a board called Raspberry Pi is used for testing the efficiency of the implemented algorithms. Raspberry Pi is derived from a hardware platform with open source project code. It includes a simple input/output (I/O) capability of the circuit board and a lot of Linux software. Raspberry Pi not only can be used to develop interactive products, such as it can read a large number of switches and the sensor signals, also can control lights, motors, and other forms of physical devices. There are two main types of Raspberry Pi including Raspberry Pi model A and model B. Model B is utilized in the experimental part. In Figure 4 the model B is illustrated.

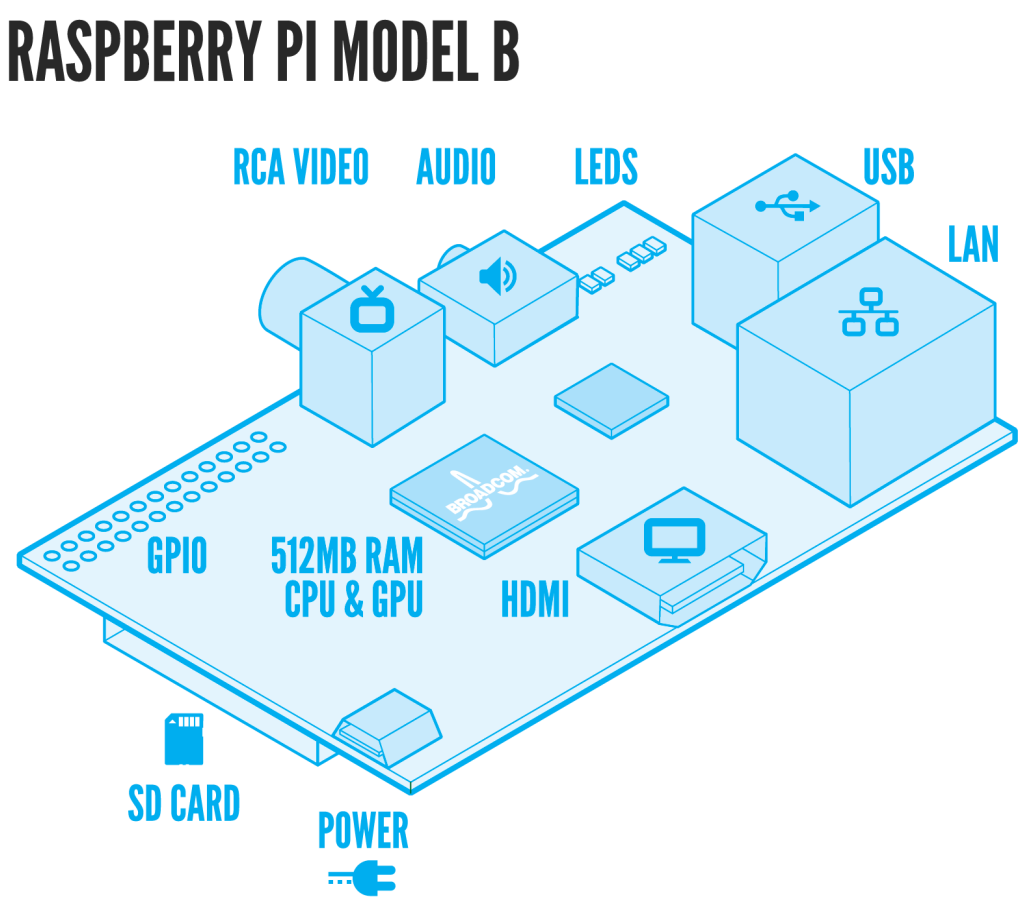

Figure 4. Raspberry Pi and its components (Burkepile 2013).

The Raspberry Pi is a credit-card size device, but it contains a 512 Megabyte (MB) of Random-Access Memory (RAM), 700 MHz Central processing Unit (CPU), two Universal Serial Bus (USB) ports and a 100mb Ethernet port. In addition to that, there are General-purpose input/output (GPIO) pins to connect some hardware. Those pins will be used in the experiment for measuring the time with the oscilloscope.

Raspberry Pi uses Raspbian operation system that based on Debian. Debian is a Linux distribution, which is under the General Public License. Linux is a clone of the Unix operation system and it has all the features that Unix has including multitask networking including IPv4 and IPv6. As a result, the Raspbian operation system is

able to make Raspberry Pi create socket communication with client. (Github; Raspbian)

2.4 The Time Measurement Functions

Since the Raspberry Pi does not contain a hardware clock on board, it uses Internet access to set the time with the help of Network Time Protocol (NTP) servers (Nprasan 2014). In the C library, the gettimeofday function can get the time for Raspberry Pi and it is not rely on updates at each timer tick (Eckhardt 1992). So this function could be used in the Raspberry Pi for calculating time.

On the other hand, CPU time is measured in clock ticks or seconds in the personal computer. The clock function in C library could be used in the code when the server is installed in PC, because this function would return the number of clock ticks elapsed since the program begins to sort the arrays (Tutorialspoint; Wikipedia 2015a).

2.5 The WiringPi Library

In the experimental part, it is to measure the time by GPIO with oscilloscope to proof that gettimeofday function is working properly for the Raspberry Pi. The Wiring Pi library is the GPIO interface library and it must be included in the program when Raspberry Pi is going to use the GPIO pins (WiringPi). There are 26 pins GPIO connector that carry signals and buses, but only 8 pins are general-purpose digital I/O pins can be programmed as digital inputs or outputs. In the experiment, one general-purpose pin and one ground pin are connected to oscilloscope in order to check gettimeofday function works properly.

2.6 Secure Shell

Secure Shell (SSH) is a cryptographic network protocol that allows a user to run commands on a machine's terminal without being closed to the machine. It is a reliable protocol that is provided for the network services and telecommunications.

SSH can effectively prevent leaking message problem as well. In the experimental part, using SSH to remotely connect with the Raspberry Pi by the PC terminal is secured. After the connection, PC terminal can safely and conveniently manipulate the Raspberry Pi (Wikipedia 2015b).

2.7 Java Client and C server

GUI usually performs through direct manipulation of graphical elements, which can display the information clearly on the computer. On the client side, the GUI is designed for generating random numbers and created socket communication with server. It is popular to use Java to design a GUI because Java is a powerful programming language that can run on all platforms support Java without the need for recompilation. Furthermore, Java also can create socket connection because it has detail knowledge that TCP is required. Now there is an application called Netbeans Integrated Development Environment (IDE) that supports Java, C/C++ and Hypertext Preprocessor (PHP) etc. This application makes the coding process much easier and simplifier because it is not necessary to manually program the basic features, which can save lots of time (Netbeans).

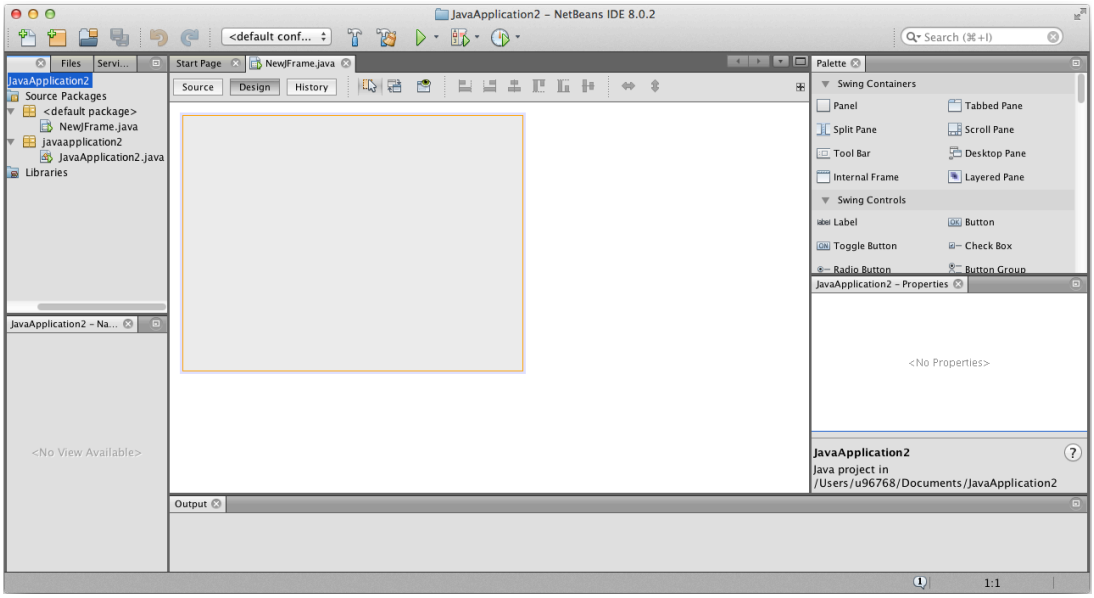

Figure 5. FrontPage of Netbeans 8.0.

Obviously, Figure 5 reveals all the common Swing containers and controls are shown at the right side of the FrontPage. Users just need to choose the components and drag them to JFrame. Clicking the source button can modify the command lines for each component.

On the other hand, C is a well-established programming language, which is a key tool in many parts of IT world. There are several reasons to choose C for the socket server on board. Firstly, Raspbian is based on the Linux operation system and C is intimately tied with Linux and Unix. Second, C with explicit memory allocation is more deterministic and the programmer can fully control over how much of memory space to be freed or allocated (Duffymo 2010). So using C to program socket server in Raspberry Pi is better to control the memory space of buffer and more suitable for Raspbian operation system.

3. SORTING ALGORITHMS

Sorting algorithms are an important part of data processing and they are widely used in many aspects, for example, computer encryption and information searches. It is an operation that makes a string of elements arranged in increased or decreased order according to the value of each element. There are many kinds of sorting algorithms and each one has its advantages and limitations. In computer science, the sorting algorithm is usually classified by the following.

1. The time complexity of calculation. It is divided into worst, average and the best performance, which is based on size n of the list. Generally, a good performance is O(n*log n), the worst case is O(n^2). An ideal sorting performance is O(n).[1]

2. Memory usage (and the use of other computer resources).

3. Stability. If element 1 and element 2 have the same value in an array, element 1 appears before element 2 in the original array, element 1 would still stay before element 2 after sorting. Then this sorting algorithm is stable. There are some naturally stable sorting methods, such as insertion sort, bubble sort, etc. and some unstable sorting algorithms, for example, heap sort, Quick sort, etc (Wikipedia 2015c).

According to the properties of different types of data, efficiency can be improved by choosing appropriate sorting algorithms. In this chapter, the concept of four sorting algorithms will be described and the time complexity of those four sorting algorithms will be analyzed as well.

3.1 Bubble Sort

Bubble sort is a simple sorting algorithm in computer science. Once this algorithm starts to sort, it will begin to scan elements repeatedly, but only compare two of them in one time. Their places would be exchanged if they were in the wrong order. After

[1] Big-Oh Notation is a mathematical way to analyze the algorithms. Definition 3.1 A function g(n) is said to be O(f(n)) is there exist constant c_0 and n_0 such that g(n) < c_0f(n) for all n > n_0. (Sedgewick 1998, P.44)

the first round, the biggest number will be show up at the end of the array. Figure 6 shows procedures of how bubble sort works in every step.

The theory of bubble sort works in the following way.

(1). Compare adjacent elements, if the first one is greater than the second, swap it.

(2). Do the same comparison as before from the first pair to the last pair. After this round, the last element should be the largest.

(3). Repeat the 2 steps above for all the elements except the last one.

(4). Repeat all the 3 steps above until there is no elements needed to swap.

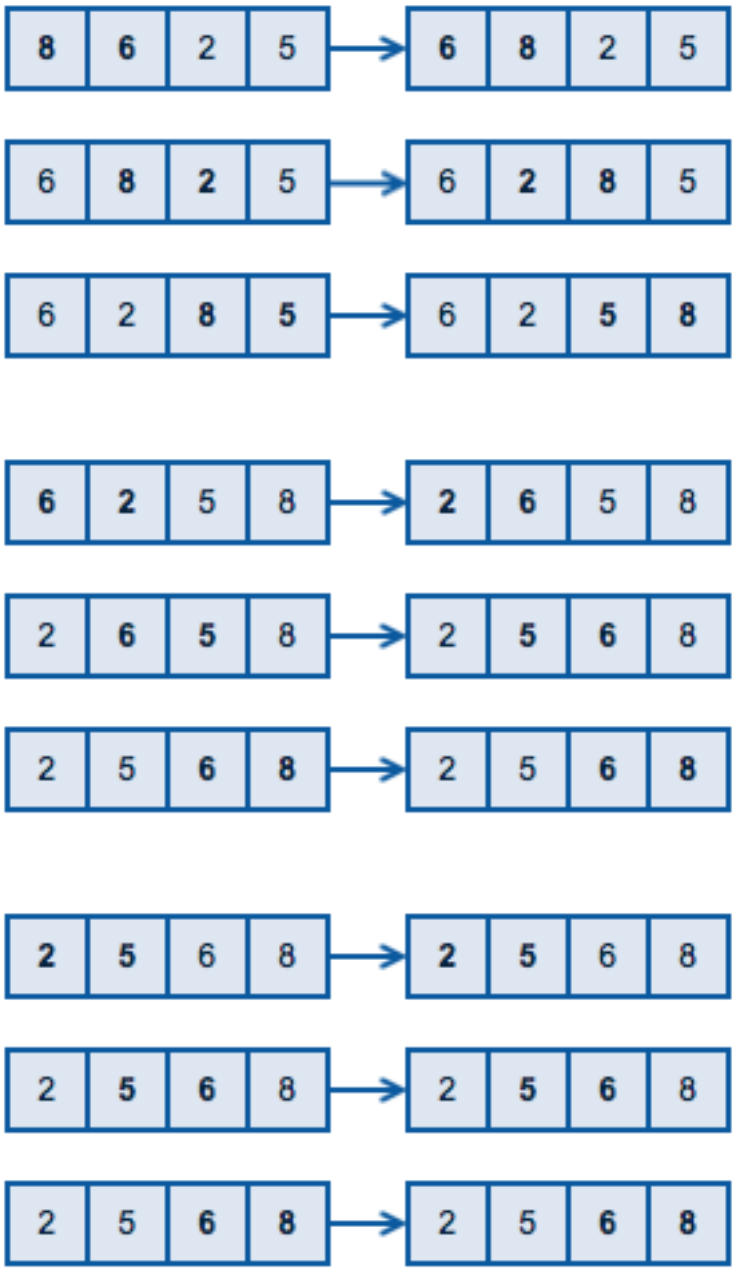

Figure 6. Flow chart of bubble sort.

The time complexity of sorting algorithm is mainly based on the complexity of the executed codes. Different sorting algorithm must have different code. The function of

bubble sort used in the experiment is shown below. The array is sorted by ascending order.

```
void bubbleSort(int *array, const int arraySize)

{

    int i, j, temp;

    for(i = 0; i < arraySize; i++)

    {

        for(j = 0; j < arraySize-1; j++)

            if(array[j] > array[j+1])

            {

                // switch variables

                temp = array[j];

                array[j] = array[j+1];

                array[j+1] = temp;

            }

    }

}
```

According to the program, let's assume execution time in each line is Cn. In the worst case, all elements in the array are in descending order (Kumar 2012).

Table 1. Execution time of bubble sort in the worst case.

Line	Time	Reputation
For loop 1	C1	n
For loop 2	C2	(n-1)
Comparison	C3	n(n-1)
Exchange	C4	(n-1)(n-i)
Initialization	K	1

According to Table 1, the total consuming time is

$$T(n) = C1*n+C2*(n-1)+C3*n*(n-1)+C4*(n-1)*(n-i)+k \quad (1)$$

$$T(n) = n^2(C3+C4) + n(C1+C2-C3-C4-i* C4) +C2+C4+k$$

$$\leq K*n^2 \text{ (when } n\geq 1, K = (C3+C4)+(C1+C2-C3-C4)+(-C4) +C2+C4+k)$$

As a result, $T(n) = O(n^2)$ in the worst case.

In the best case of bubble sort, the array should be already sorted. In this case, there should be a check command added in for loop in order to get the best performance of bubble sort. This check command can avoid for loop goes to n times, n is the size of the array.

```
void bubbleSort(int *array, const int arraySize)

{

int i, j, temp;

int swap = 0;

for(i = 0; i < arraySize; i++)

{
```

```
swap = 0;
            for(j = 0; j < arraySize-1; j++)
                  if(array[j] > array[j+1])
                  {
                        // switch variables
                        temp = array[j];
                        array[j] = array[j+1];
                        array[j+1] = temp;
                        swap = swap + 1;
}
            if(swap == 0)
            {
              i = j = arraySize;
              break;
            }
      }
}
```

According to this upgrade function, there are (n-1) times comparison, 0 times exchange and n times to check if all the elements are sorted or not.

Table 2. Execution time of bubble sort in the best case.

Line	Time	Reputation
For loop 1	C1	n
For loop 2	C2	(n-1)
Comparison	C3	(n-1)
Exchange	C4	0
Check	C5	n
Initialization	K	1

According to the Table 2, the time complexity of bubble sort should be

$$T(n) = C1*n + C2*(n-1) + C3*(n-1) + C4*0 + C5*n + k \quad (2)$$

$$= n*(C1+C2+C3+C5) + (-C2-C3+k)$$

$$\leq K*n \quad (\text{when } n \geq 1, K = (C1+C2+C3+C5) + (-C2-C3+k))$$

The complexity of bubble sort in the best case is T(n) = O(n).

3.2 Heap Sort

3.2.1 Tree and Heap

A tree is a collection of nodes, the collection can be empty or consist of a node r without parents and zero or other sub trees T1, T2, ... , Tk. This node is called root, those sub-trees' nodes are connected by a directed edge to node r. The root of every sub-tree is a child of node r. Node r is parent of each sub-tree root. Nodes with same parent are siblings. A node without children is called leave. Figure 7 shows a binary tree whose node cannot have more than two children.

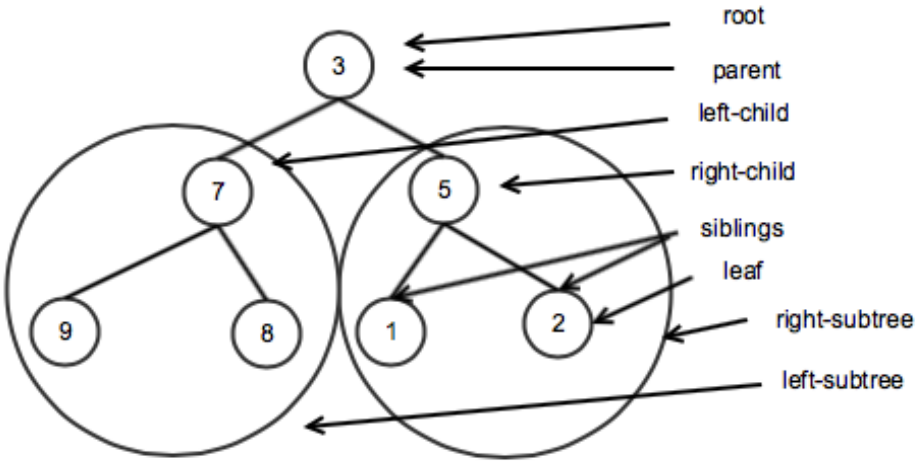

Figure 7. A tree.

A heap is always a complete binary tree. Usually, heap is regarded as elements of an array group as a tree. Heap is always satisfied with the properties below.

1. Each node's value is not greater than the value of their parent.

2. A heap should be a complete binary tree, which means its outline is a triangle (there might be a missing chunk on the lower right possibly).

According to the properties, it is easy to find out the relationship between elements position in the tree and elements position in the array. For any element position i in the array, the left child is in the location 2i, the right child is in the cell next to the left child (2i+1) and the parent is in the position $\lfloor i/2 \rfloor$ (Weiss, 1994).

Heaps can be classified as a "max heap" or a "min heap". A max heap should have a root node which contain the highest value of all the nodes and values of all the parent nodes are not less than the values of children, while a min heap should have a root node which contain the lowest value of all the nodes and values of all the parent nodes are not greater than the values of children. If an array is a heap, the key of the

root is the greatest or the least in this array. Figure 8 shows the differences between max heap and min heap.

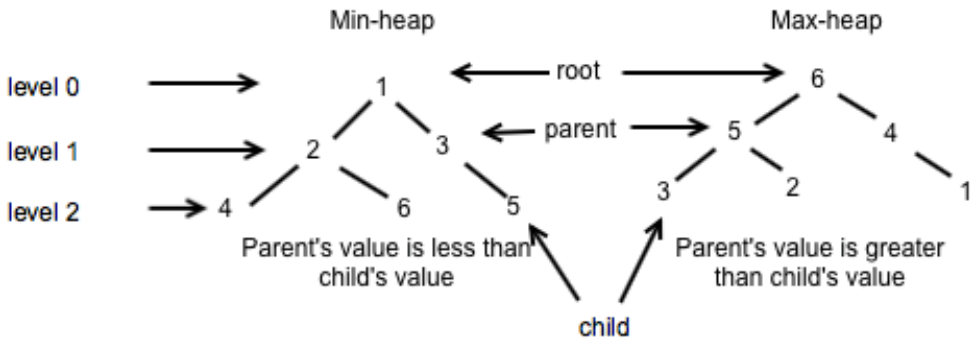

Figure 8. Min-heap and Max-heap.

If array should be sorted in ascending order, max-heap would be utilized in the heap sort. Otherwise, min-heap would be used in the heap sort for descending order.

3.2.2 Heap Sort

Heap sort is a sorting algorithm based on heap (or tree) data structure. It utilizes the property of max heap (or min heap), which means the root of the heap is the greatest or the least. It is much easier to pick up the element with highest or the lowest value in an array without order. Steps of heap sort are divided into two parts, which includes build_heap and heapify.

Heapify is the key in the heapsort. The root contains the max value needs to be taken out and store the element in a safe place in the array. Then the heapsize is decreasing and the root is empty. The left and right sub-trees below the root would still be heaps. So generally, the responsibility of heapify is to check the first heap property, which is swap the larger child up if it is satisfied with the heap property and to repeat this

recursively to the lower levels. In detail, the heapify section includes n successive inserts. In the insert routines, percolation is implemented for repeating swaps until the tree is in the correct order (Weiss, 1994). Inserting an element X into the heap needs to create a hole in the next available location first. If X is placed in the right position, insertion of element X is finished. Otherwise X needs to be exchanged the position with its parent in the heap until X is satisfied with the heap properties.

In the coding structure, heapify should have five main steps which are mentioned below (Moore 1999).

```
HEAPIFY(A,i)

 1 left <- left(i)

 2 right <- right(i)

 3 if node i has a larger child

 4       then swap A[i] and largest child

 5               HEAPIFY(A,location largest child was in)
```

Build_heap is collecting n elements and placing them into an empty heap. To build the heap, which needs to call heapify again and again and begin with the lowest parents to make sure every node position is satisfied with the heap properties. In the build_heap coding, there should be three main steps.

```
BUILDHEAP(A)

 1 heapsize <- length(A)

 2 for i <- floor(length(A)/2) downto 1    **from lowest parent up to root**

 3         do HEAPIFY(A,i)
```

First, build the heap according to the rules of build_heap. Figure 9 – Figure 15 show the steps of build_heap.

Second, heapify. Remove the root and exchange this root element with the last element then insert this last element in a proper position in the heap according to the heap properties. Repeat this step until all elements are sorted. Figure 16 – Figure 35 are shown the details of heapify.

When those two parts are finished, the result should display a sorted array.

In the coding system, it is expressed similarly as below.

```
HEAPSORT(A)

 1 BUILDHEAP(A)

 2 for i <- length(A) downto 2    **start at bottom and work up**

 3         do swap A[1] and A[i]

 4              heapsize <- heapsize – 1

 5              HEAPIFY(A,1)
```

For example, there is an array: 25,29,20,17,27,26.

Part 1 Build_heap

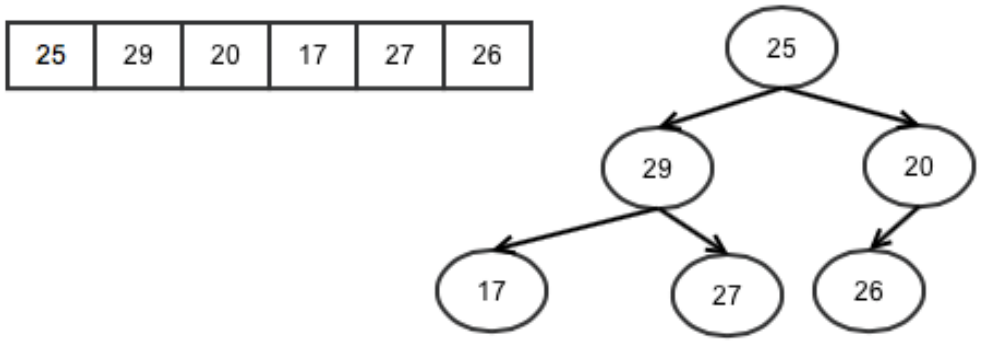

Figure 9. Original tree.

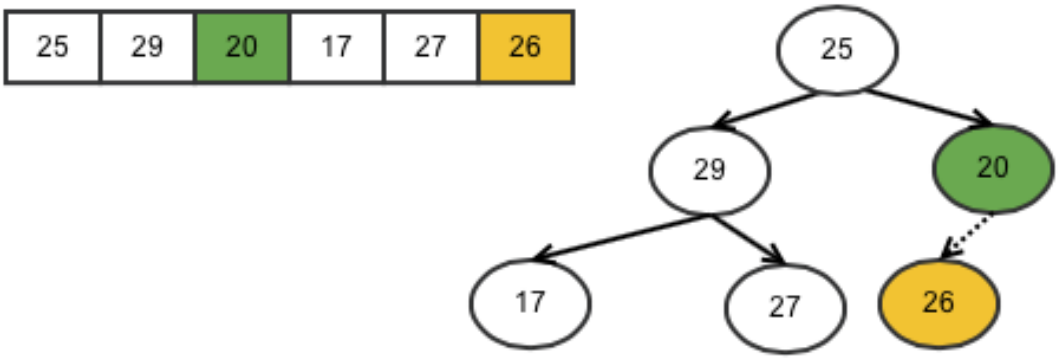

Figure 10. Step 1 of build_heap.

Start with the rightmost node at height 1 - the node at position 3 = Size/2.

It has one greater child and has to be percolated down.

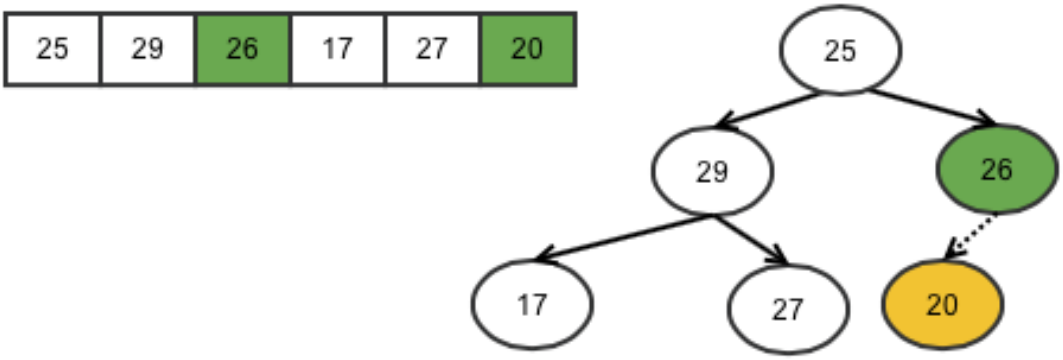

Figure 11. Step 2 of build_heap: exchange array[2] content with array[5].

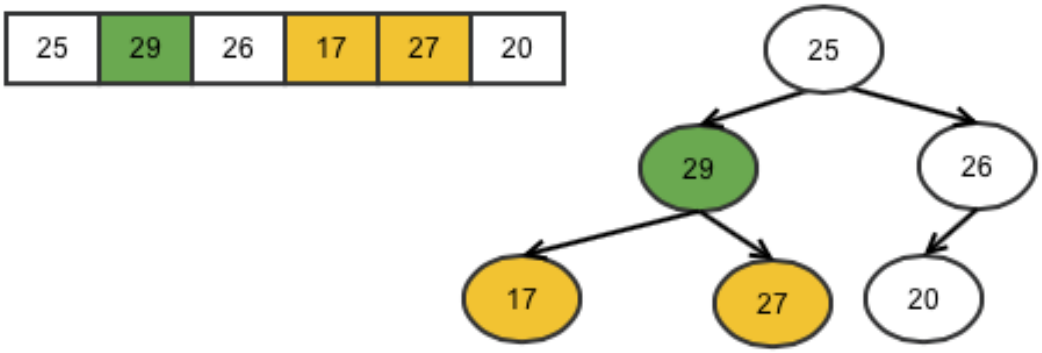

Figure 12. Step 3 of build_heap: children of array[2] are smaller than 29, no percolation is needed.

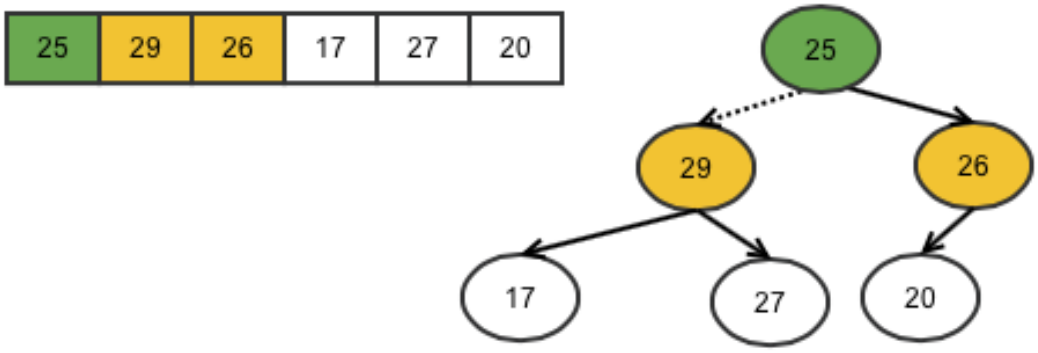

Figure 13. Step 4 of build_heap: The last node to be processed is array[0]. Its left child is greater than the right one. The content of array[0] has to be percolated down to the left, so swapped with array[1].

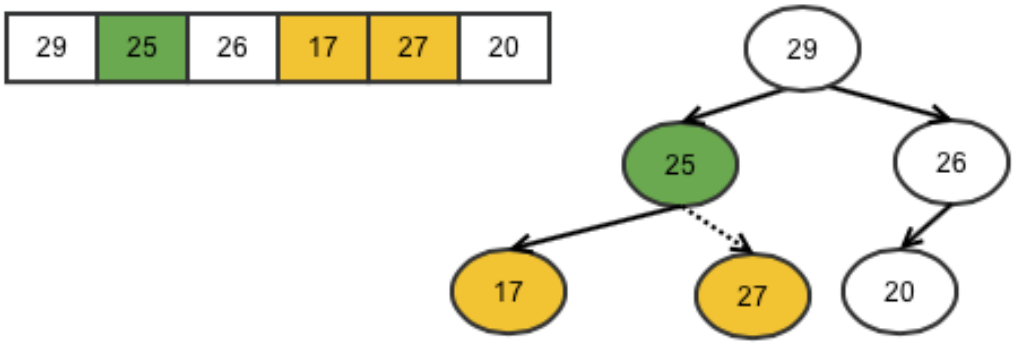

Figure 14. Step 5 of build_heap:the children of array[2] are greater, so item 25 has to be moved down further, swapped with array[5].

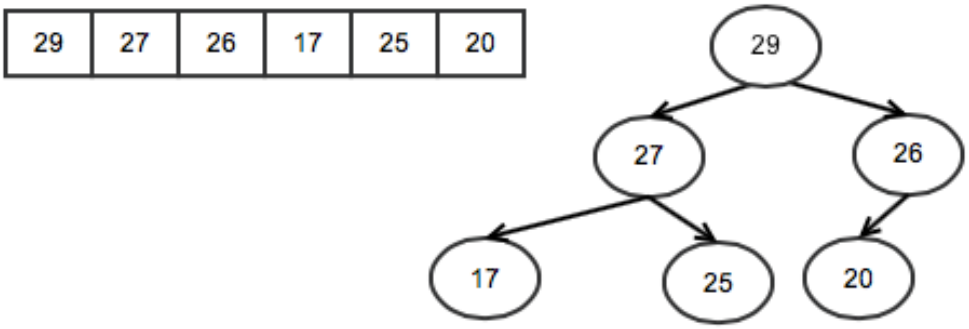

Figure 15. Build_heap is finished.

Part 2 Heapify

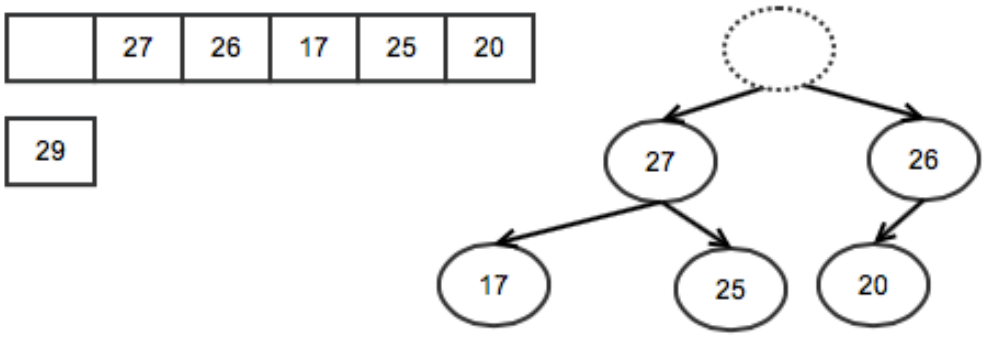

Figure 16. Delete the root element 29 and store it in a temporary place. A hole will be created at the top.

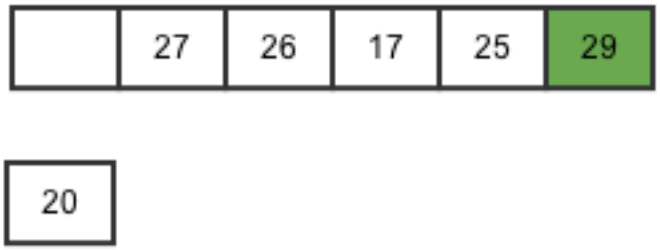

Figure 17. Swap 29 with the last element of the heap. As 20 will be adjusted in the heap, its cell will not be a part of heap. It becomes a cell from the sorted array instead.

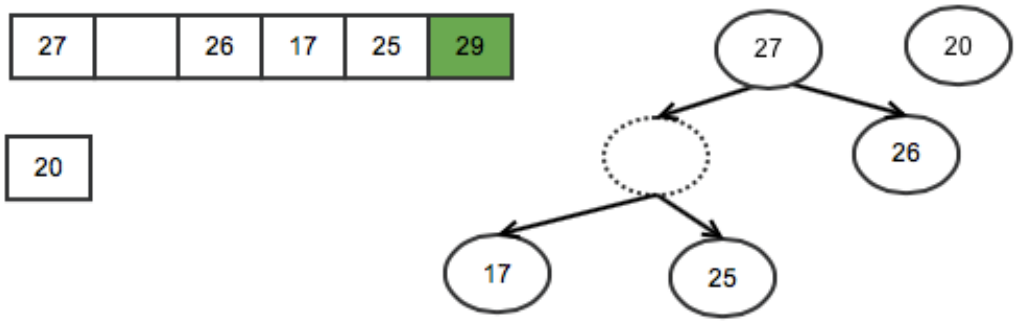

Figure 18. Percolate down the hole.

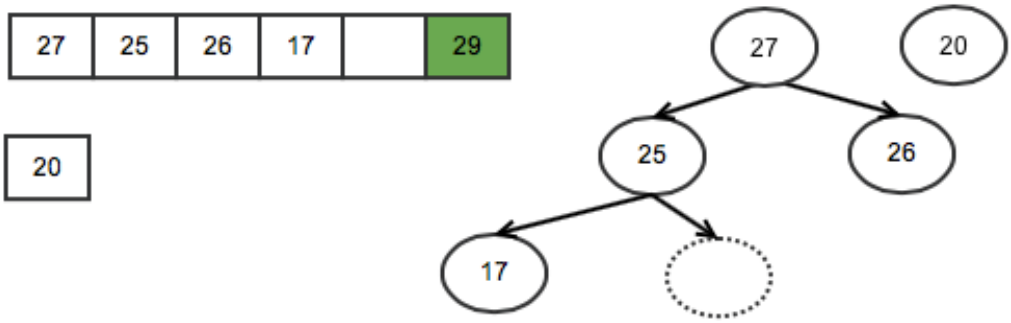

Figure 19. Percolate one more time as 20 is smaller than 25, it cannot be inserted in the previous hole.

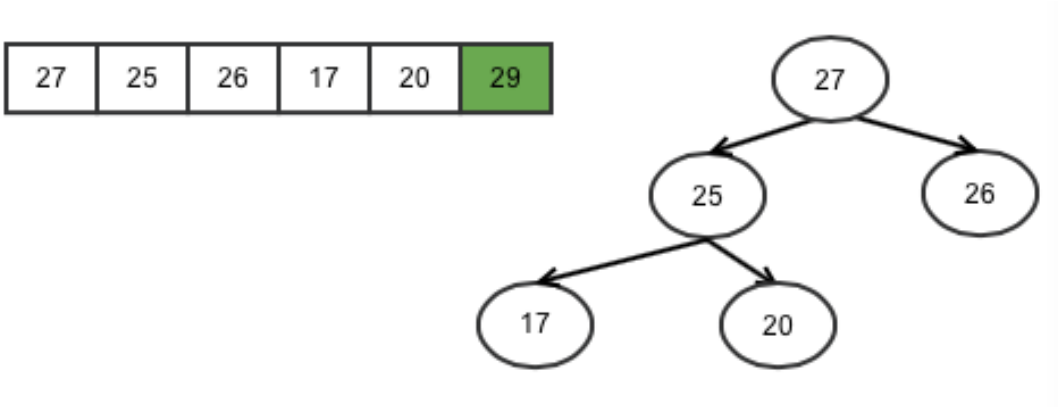

Figure 20. Insert 20 into the hole.

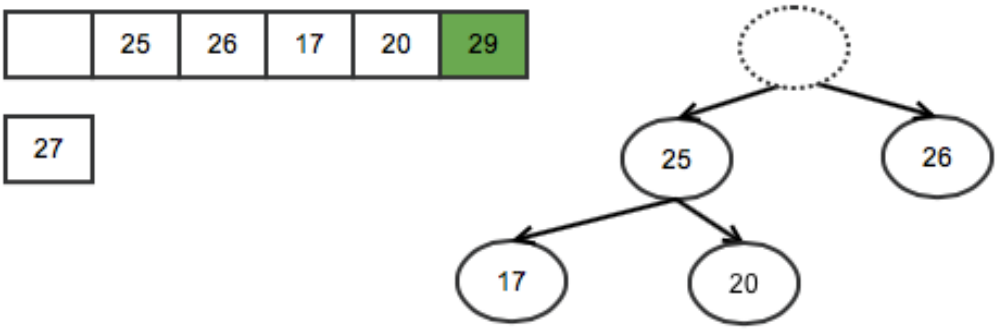

Figure 21. Remove the root element 27 and store it in a temporary place. A hole is created at the root.

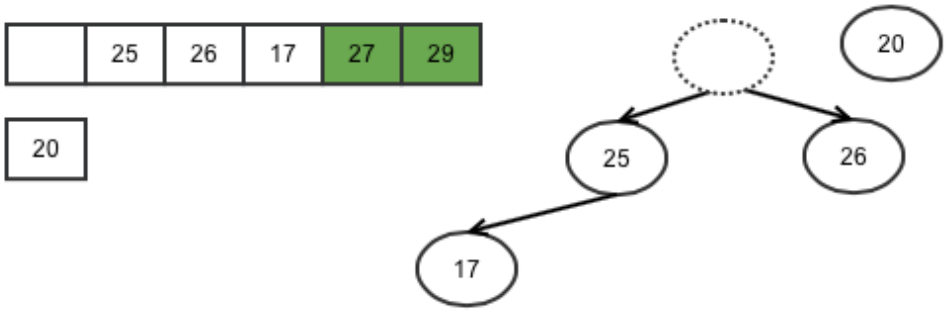

Figure 22. Swap 27 with the last element of the heap. As 20 will be adjusted in the heap, its cell will no longer be a part of the heap. It becomes a cell instead of the sorted array.

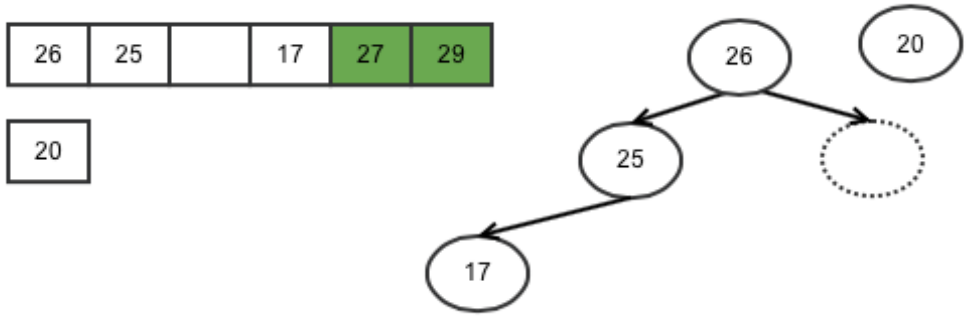

Figure 23. The element 20 is less than the children of the hole, and the hole is percolated down.

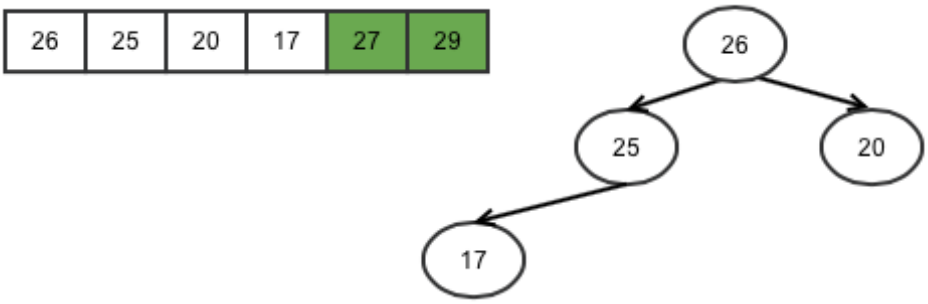

Figure 24. Insert 20 in the hole.

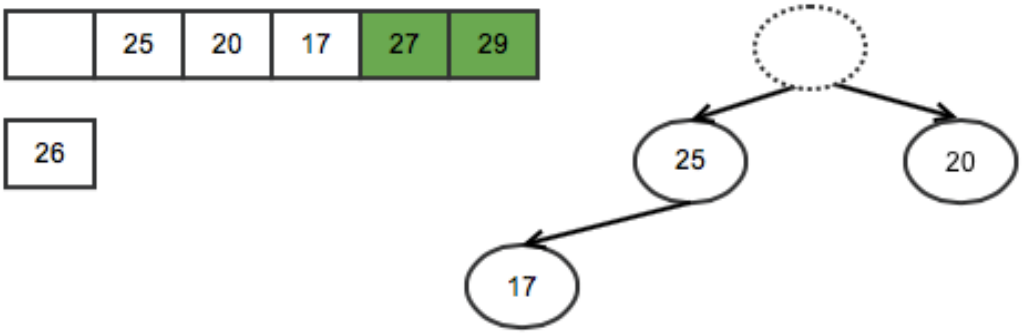

Figure 25. Store 26 in a temporary place. A hole is created at the top.

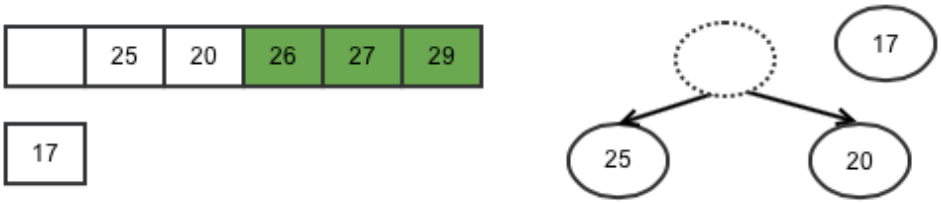

Figure 26. Swap 26 with the last element of the heap. As 17 will be adjusted in the heap, its cell will no longer be a part of the heap. It becomes a cell instead of the sorted array.

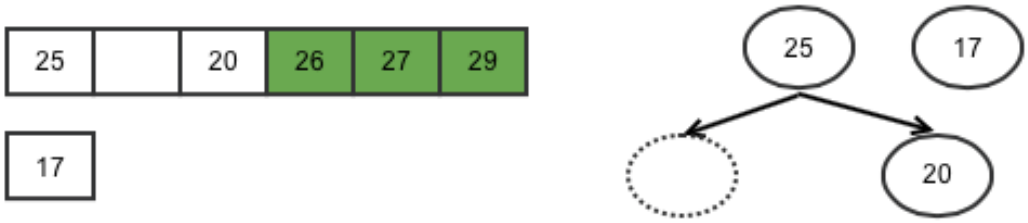

Figure 27. Percolate the hole down since 17 cannot be inserted there. It is less than the children of the hole.

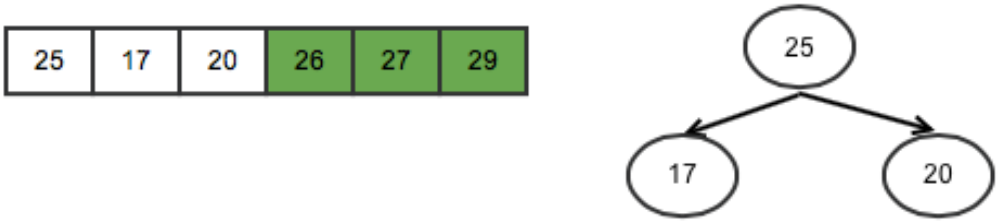

Figure 28. Insert 17 in the hole.

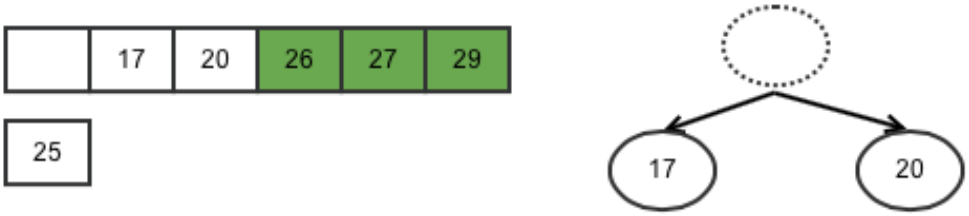

Figure 29. Remove the root element 25. Store 15 in a temporary location. A hole is created.

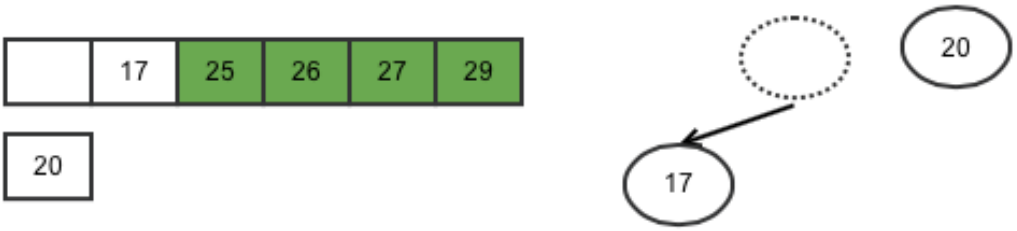

Figure 30. Swap 25 with the last element of the heap. As 20 will be adjusted in the heap, its cell will no longer be a part of the heap. It becomes a position from the sorted array.

Figure 31. Store 20 in the hole (20 is greater than the children of the hole).

Figure 32. Remove 20 from the heap and store it into a temporary location.

Figure 33. Swap 20 with the last element of the heap. As 17 will be adjusted in the heap, its cell will no longer be a part of the heap. It becomes a cell from the sorted array.

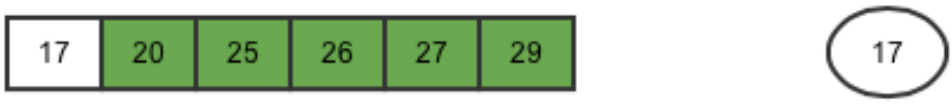

Figure 34. Store 17 in the hole as the only remaining element in the heap.

Figure 35. 17 is the last element from the heap. The array is sorted completely.

As it is mentioned before, the code of the heap sort is divided into three parts.

1. Build max heap[x].
2. For loop to heapify.
3. Heapify.

When it is written down as a program, it should be similar with this as following.

```
void heap_sort(int *array, int number)

{

    int start, end;

    /* build-max-heap */

    for (start = (number-2)/2; start >=0; start--) {

        heapify(array, start, number);

    }



    for (end=number-1; end > 0; end--) {

        do{

            int t=array[end]; array[end] =array[0]; array[0]=t;
```

```
            }
            while(0);
            heapify (array, 0, end);
        }
}
void heapify (int *array, int start, int end)
{
        int root = start;
        while ( root*2+1 < end ) {
            int child = 2*root + 1;
            if ((child + 1 < end) && (array[child]<array[child+1])) {
                child += 1;
            }
            if (array[root]< array[child]) {
                do {
                    int t=array[child]; array[child]=array[root]; array[root]=t;
                } while (0);
                root = child;
```

```
            }
            else
                return;
        }
}
```

In this program, there are two main functions: build_heap and heapify. Therefore total time T(n) can be computed in the following way.

T(n) = $T_{build-heap}$(n) + $T_{heapify}$(n) * for loop times (3)

For the execution time of build-max-heap, let's assume n is the total number of all the elements. Because the heap is a complete binary tree, let's suppose n = 2^{h+1}-1, where h is the height of the tree.

According to this assumption, the tree will have h levels. Level 0 has 1 node, level 1 has 2 nodes, and up to level h has 2^h nodes. It is shown below in Figure 36.

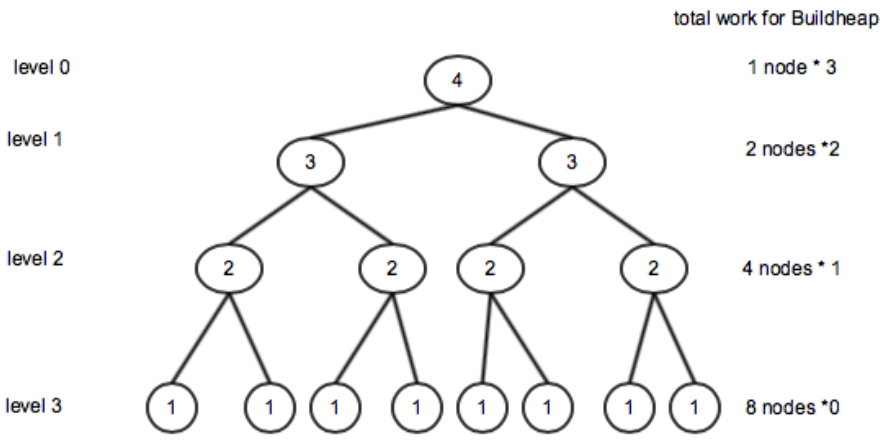

Figure 36. Total build_heap work.

Therefore, there are 2^h nodes on the bottommost level, but there is no heapify work in this level. At the second bottommost level there are 2^{h-1} nodes, each of them might percolate down 1 level. At the third bottommost level there are 2^{h-2} nodes, each of them might percolate down 2 levels. So in generally, there are 2^{h-j} nodes at the j th bottommost level, each of them might percolate down j level (Heaps). So the total work for build heap is

$$T_{build-heap}(n) = \sum_{j=0}^{h} j * 2^{h-j} \quad (4)$$

$$= \sum_{j=0}^{h} j * \frac{2^h}{2^j}$$

[2] $$= 2^h \sum_{j=0}^{h} \frac{j}{2^j} \leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \leq 2^h * 2 = 2^{h+1}$$

Now recall the equation at the beginning, 2^{h+1}-1=n, the total working time of build-heap is T(n) $\leq$ n +1 $\leq$ Kn (when n$\geq$1, K = 2). As a result, T (n) = O (n).

For the heapify running time, it is recursively called h times at most, where h is the height of the sub-tree starting at node i. Every time a percolate down takes c steps, for example, exchange the element of the nodes, where c is some constant.

So for all the elements, there are c*h steps at most. Therefore, the running time of percolate down in the worst case is O(h). Since the relation between the height of the tree h and the size of the array n is 2^{h+1}-1=n, the time complexity of heapify is O(log(n-1)-1), which could be simplified as O(log n).

According to the for loop in the code, it is from n-1 down to 1. So there are n-1 times heapify happens in the heap sort.

---

[2] Here is an infinite general geometric series, for any constant x < 1.

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}$$

let's take the derivative of both side with respect to x, and multiply by x :

$$\sum_{j=0}^{\infty} jx^{j-1} = \frac{1}{(1-x)^2} \qquad \sum_{j=0}^{\infty} jx^j = \frac{x}{(1-x)^2}$$

when x = $\frac{1}{2}$, then

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$$

As a result, the total working time for the heap sort is

$$T(n) = O(n) + O((n-1)\log n) \quad (5)$$

So $T(n) = O(n \log n)$

Therefore, heap sort takes O(n log n) time in the worst case and in the best case the O(n log n) is the best possible lower bound (Moore 1998).

3.3 Insertion Sort

Insertion sort is a simple implemented sorting algorithm. The method is to consider one element at a time, inserting each into the proper position among the elements that have been sorted. In the best case, the time complexity is O(n), while it takes $O(n^2)$ time in the worst case. The average performance is $O(n^2)$. Generally, insertion sort includes the following procedures. Figure 37 shows an example of insertion sort.

1. Assume the first item is already sorted.

2. Check the next new item of the array and compare it with the element before it, then insert it into to a proper position. Repeat this step until all the elements are sorted already.

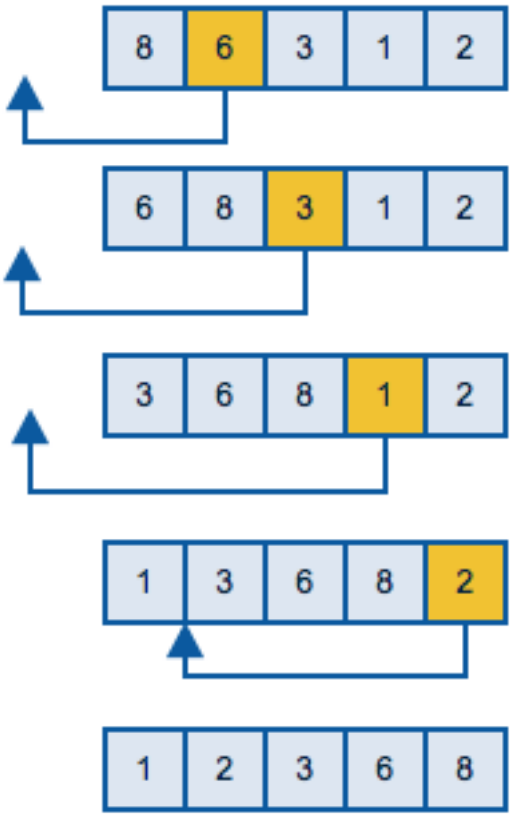

Figure 37. Flow chart of insertion sort.

The function of insertion sort used in the experiment is shown below. The array is sorted by ascending order.

```
void insertionsort(int *array, int number)

{

    int i,j;

    int temp;

    for (i = 1; i <= number-1; i++)
```

```
    {
        temp = array[i];
        j = i;
        while (j > 0 && temp<array[j-1]) {
            array[j] = array [j-1];
            j--;
        }
        array[j] = temp;
}
}
```

As for the complexity of insertion sort, Big-Oh notation of the insertion sort is going to be analyzed.

According to the code, if the array is in descending order, there must be i+1 times comparison and i times exchange in each for loop in the code. So the iteration of each command line is shown below (Kumar 2012).

Table 3. Execution time of insertion sort in the worst case.

Line	Time	Reputation
Initialization	k	1
For loop	C1	n-1

Assignment	C2	n-1
While loop	C3	$\sum_{j=2}^{n} j$
Shift	C4	$\sum_{j=1}^{n-1} j$
j--	C5	$\sum_{j=1}^{n-1} j$
Exchange	C6	n-1

According to Table 3, the total time consumption can be computed in the following way.

$$T(n) \text{ [3]} = k + C1*(n-1) + C2*(n-1) + C3*(\frac{n(n+1)}{2} - 1) + C4*\frac{n(n-1)}{2} + C5*\frac{n(n-1)}{2} + C6*(n-1) \quad (6)$$

$$= k + C1*n - C1 + C2*n - C2 + \frac{1}{2}*C3*n^2 + \frac{1}{2}*C3*n + \frac{1}{2}*C4*n^2 - \frac{1}{2}*C4*n + \frac{1}{2}*C5*n^2 - \frac{1}{2}*C5*n + C6*n - C6$$

$$= n^2*(\frac{1}{2}*C3 + \frac{1}{2}*C4 + \frac{1}{2}*C5) + n*(C1 + C2 + \frac{1}{2}*C3 + \frac{1}{2}*C4 + \frac{1}{2}*C5 + C6) + (k - C1 - C2 - C6)$$

$$\leq K*n^2 \quad (\text{when } n \geq 1, K = (\frac{1}{2}*C3 + \frac{1}{2}*C4 + \frac{1}{2}*C5) + (C1 + C2 + \frac{1}{2}*C3 + \frac{1}{2}*C4 + \frac{1}{2}*C5 + C6) + (k - C1 - C2 - C6))$$

The complexity of the insertion sort is $O(n^2)$ in the worst case.

While the insertion sort is in the best case, there should not be any exchange in a sorted array. So the iteration of each command line is shown as following.

[3] Equation: $\sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$

Table 4. Execution time of insertion sort in the best case.

Line	Time	Reputation
Initialization	k	1
For loop	C1	n-1
Assignment	C2	n-1
While loop	C3	n-1
Shift	C4	0
j--	C5	0
Exchange	C6	n-1

According to Table 4, the total execution time of insertion sort in the best case is

$$T(n) = k + C1*(n-1) + C2*(n-1) + C3*(n-1) + C6*(n-1) \quad (7)$$

$$= n*(C1+ C2+ C3 + C6) + (k - C1- C2 - C3 – C6)$$

$$\leq K* n \text{ (when } n \geq 1, K = (C1+ C2+ C3 + C6) + (k - C1- C2 - C3 – C6))$$

$$= O(n)$$

In the best case, complexity of insertion sort is O(n).

3.4 Quick Sort

Quicksort is regarded as the most excellent sorting algorithm in practice so far. Even though it has the worst case running time of $O(n^2)$, the average running time of quicksort is O(n*log n) as its inner loop is highly optimized. Quicksort is a divide-and-conquer recursive algorithm. The basic quicksort consists of four procedures, which are shown below.

1. Pick a pivot from the elements of the array.
2. Sort this array, if the element is larger than the pivot, then this element comes after the pivot; while the element is less than the pivot then it comes before the pivot. After this partition, the pivot is in its final position.
3. Repeat those two steps to the sub-arrays until every element is in the right order.

Figure 38 shows a simple example of quicksort. The array elements are 15,10,8,20,23,2,16.

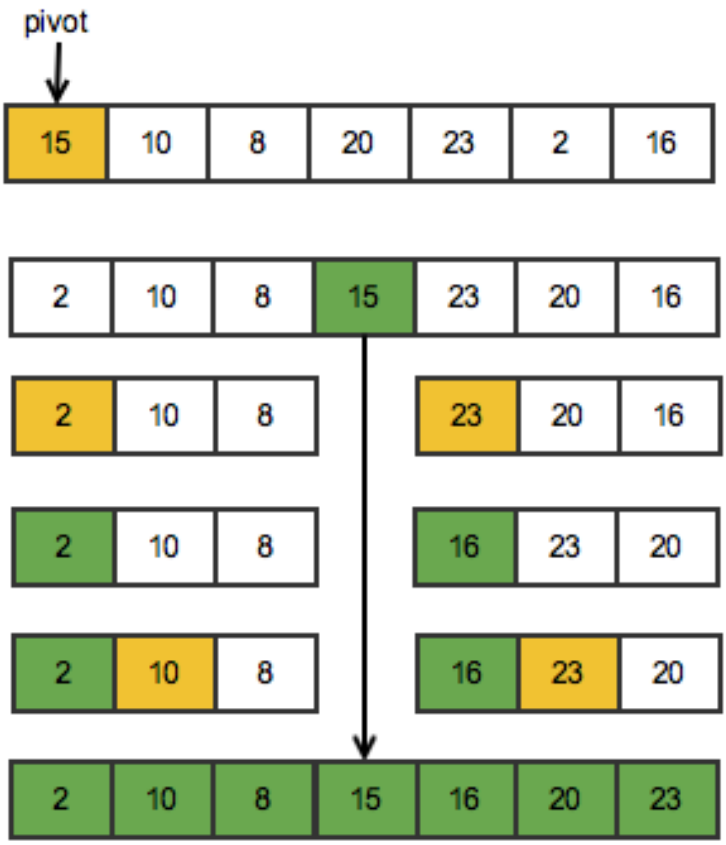

Figure 38. Flow chart of quick sort.

The function of quick sort (in C) used in the experiment is shown below. The array is sorted by ascending order.

```
void quicksort(int *array, int first, int last)

{

    int pivot, j, temp, i;

    if (first < last){

        pivot = first;

        i = first;

        j = last;

        while (i < j){

            while (array[i] <= array[pivot] && i<last)

                i++;

            while (array[j]>array[pivot])

                j--;

            if (i < j){

                temp = array[i];

                array[i] = array[j];

                array[j] = temp;

            }

        }
```

```
            temp = array[pivot];

            array[pivot] = array[j];

            array[j] = temp;

            quicksort(array, first, j - 1);

            quicksort(array, j + 1, last);

        }

}
```

According to the program of quick sort, the elements are smaller than the pivot will be appeared to the left of the pivot, and all the elements are greater than the pivot arranged to the right of the pivot. Let's suppose that the chosen pivot has divided the array into two parts. There are four steps of partitioning considered (Quicksort Analysis).

1. Traverse the original array once and copy all the elements less than the pivot into a temporary array.
2. Copy the pivot into the temporary array.
3. Copy all the elements greater than the pivot into the temporary array.
4. Transfer all the elements from the temporary array into original array.

Those four steps will take 4n operations. But this is just one of the partitioning methods, let's assume partition time would take α*n. Now assume one array size is k, the other size is n-k. So the time consuming equation is

$$T(n) = T(k) + T(n - k) + \alpha * n \quad (8)$$

T(n) refers to the time consumption of sorting n elements.

In the worst case, the pivot would be the greatest element of the array, which means the pivot is in the first place of a descending array and the rest of the elements are less than the pivot. So k = 1 and n-k = n-1.

$$T(n) = T(1) + T(n-1) + \alpha * n \quad (9)$$

In the sub-array sorting, the same situation will be continued. The second greatest one is in the first place as pivot.

$$T(n) = T(n-1) + T(1) + \alpha * n \quad (10)$$

$$= [T(n-2) + T(1) + \alpha*(n-1)] + T(1) + \alpha*n$$

Now it is easy to find out this complexity equation will be recursive.

$$T(n) = T(n-2) + 2T(1) + \alpha*(n-1+n) \quad (11)$$

$$= [T(n-3) + T(1) + \alpha*(n-2)] + 2T(1) + \alpha*(n-1+n)$$

$$= T(n-i)+iT(1)+\alpha(n-i+1+.....+n-2+n-1+n)$$

$$= T(n-i) + iT(1) +\alpha\left(\sum_{j=0}^{i-1}(n-j)\right)$$

According to the code, there would be n-1 times to repeat. So i = n-1.

$$T(n) = T(1) + (n-1)T(1) +\alpha\left(\sum_{j=0}^{n-2}(n-j)\right) \quad (12)$$

$$= n*T(1)+\alpha(n(n-2)-(n-2)(n-1)/2)$$

$$\leq K* n^2$$

So in worst case, the complexity of quick sort is $O(n^2)$. This result is also the same when the array is ascending. (Quicksort Analysis)

While in the best case, each step of pivot is always supposed to divide the array or sub-array into equal parts. The execution time will be

$$T(n) = 2T(n/2) + \alpha n \quad (13)$$

$= 2(2T(n/4) + \alpha n/2) + \alpha n$

$= 2^2 T(n/4) + 2\alpha n$

$= 2^2(2T(n/8) + \alpha n/4) + 2\alpha n$

$= 2^3 T(n/8) + 3\alpha n$

$= 2^k T(n/2^k) + k\alpha n$

The recursive sorting will continue until $n/2^k \leq 1$. Let's put the k = log n in the equation,

$T(n) = n*T(1) + \alpha n \log n$ (14)

$\leq K*(n \log n)$

So the best case for T(n) is O(n log n).

4. EXPERIMENTAL PART

In the experimental part, the content is to evaluate and compare the time consumption of the four different sorting algorithms, introduced in chapter 3, with PC and Raspberry Pi.

4.1 Hardware for Simulations

As it has been mentioned before, a PC and an Embedded PC (Raspberry Pi) are used to evaluate the introduced sorting algorithms. The PC consist of an Intel Core I5 processor with 4 GB of Random Access Memory (RAM). The Raspberry PI consists of a 700MHz ARM11 processor with 512MB Synchronous Dynamic Random-access Memory (SDRAM). In additon there are GPIO pins. (Embedded L.W., 2015.)

4.2 Software Implementation

To evaluate the algortihms a program must be written that contains the implementation of the algorithms. As a requirement, the program with the algortihms should run on different devices, as well as on devices without graphic port. Furthermore the program should be easily extendable. These requirements lead to the decision of implementing a client server based program. The server contains the implemented algortihms while the client is a GUI responsible for generating random numbers which are sent to the server where they are going to be sorted according a selected sorting algorithm. After the numbers have been sorted they are sent back to the GUI where they are displayed. Also the sorting time will be sent to the GUI.

The client GUI is programmed with Java using the Netbeans 8.0 development environment. Figure 39 illustrates the GUI (client software).

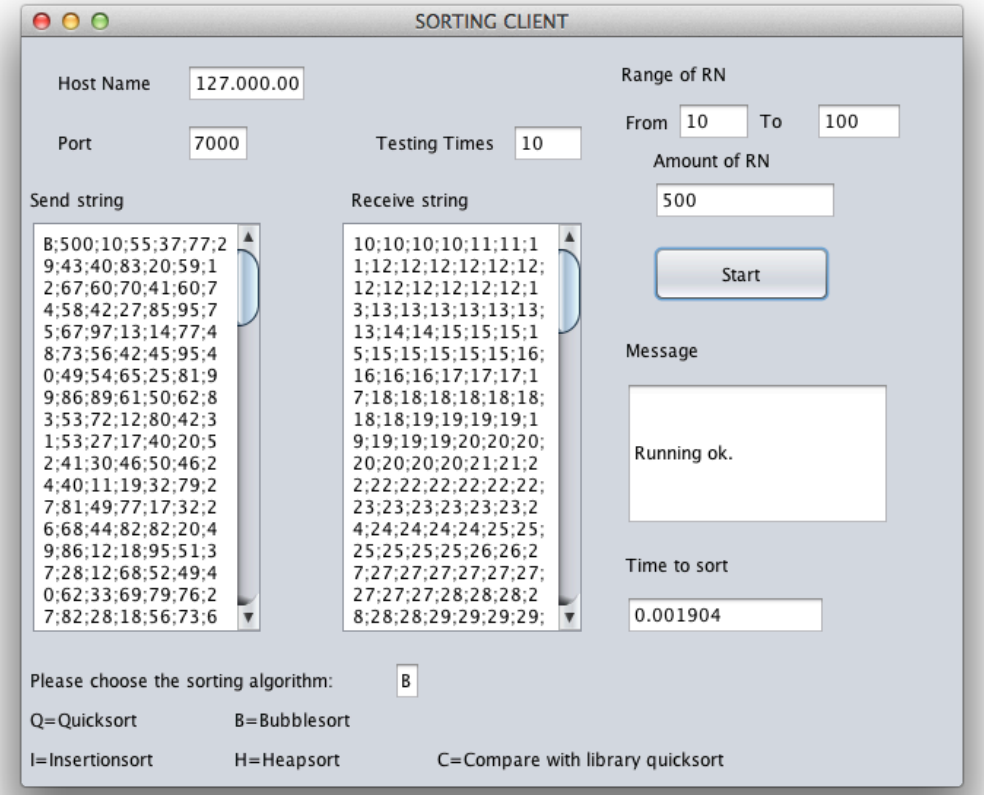

Figure 39. The Client GUI.

In this GUI, the user can type in the correspoding host name and port number of the server in the text field. Furthermore the testing times, range and amount of random numbers can be entered. The bottommost textfield is for user to choose the sorting algorithm. After pressing the start button, the client GUI will generate the uniformly distributed random numbers, display them and send them to the corresponding server together with the selected character for a certain sorting algorithm. The GUI waits until it receives the sorted numbers with the sorting time from the server. Then it displays the results. If the testing times field is initialized with a value bigger than one then new random numbers are generated and sorted until the number of iterations is equal to the value of the testing times field. The average sorting time will be displayed on the GUI.

By clicking on the left-up corner cross button, the user closes the application and also the socket connection.

4.3 Test Environment

To evaluate the sorting algortihms on the PC and on the Raspberry Pi the following setups had to be done. For the performance evaluation of the algortihms on the PC, the client as well as the server were running on the same PC. The server ran on the localhost (IP Address: 127.0.0.1) to whom the client established its connection.

For the performance evaluation of the algortihms on the PC, the server ran on the Raspberry Pi and the client on the PC. The PC was connected with the raspberry Pi over Ethernet. Figure 40 shows how the raspberry PI was connected with the PC. The Raspberry Pi was connected with a wireless router via an Ethernet cable. Furthermore, the PC was connected with the wireless router, wirelessly. Both, PC and Raspberry Pi received their IP Address over Dynamic Host Configuration Protocol (DHCP). To establish a connection to the server running on the Raspberry Pi, the IP Address which was assigned to the Raspberry Pi from the wireless router had to be entered in the Server IP textfield of the GUI. The used port number was 7000.

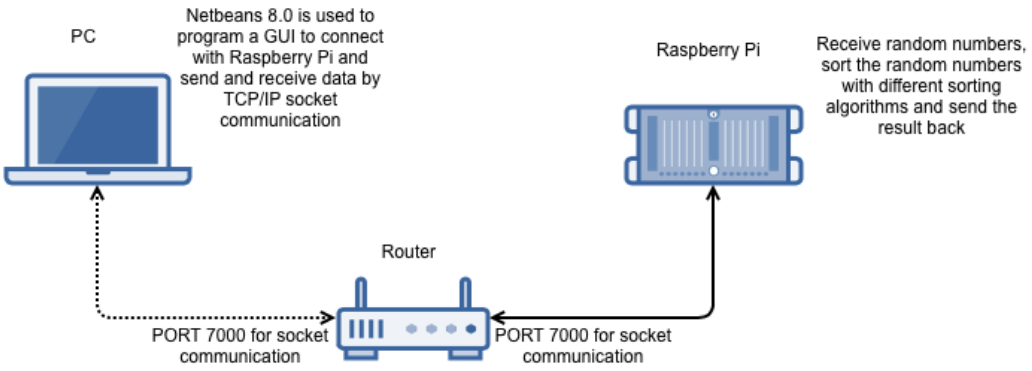

Figure 40. The Implementation of Devices Connection.

In this experiment, the used PC was a MacBook Pro (13-inch, late 2011) and its operation system is OS X version 10.9.5. SSH (short for Secure Shell) was utilized for

logging in the raspberry Pi. After logging in the Raspberry Pi in the terminal in PC, the server could be executed.

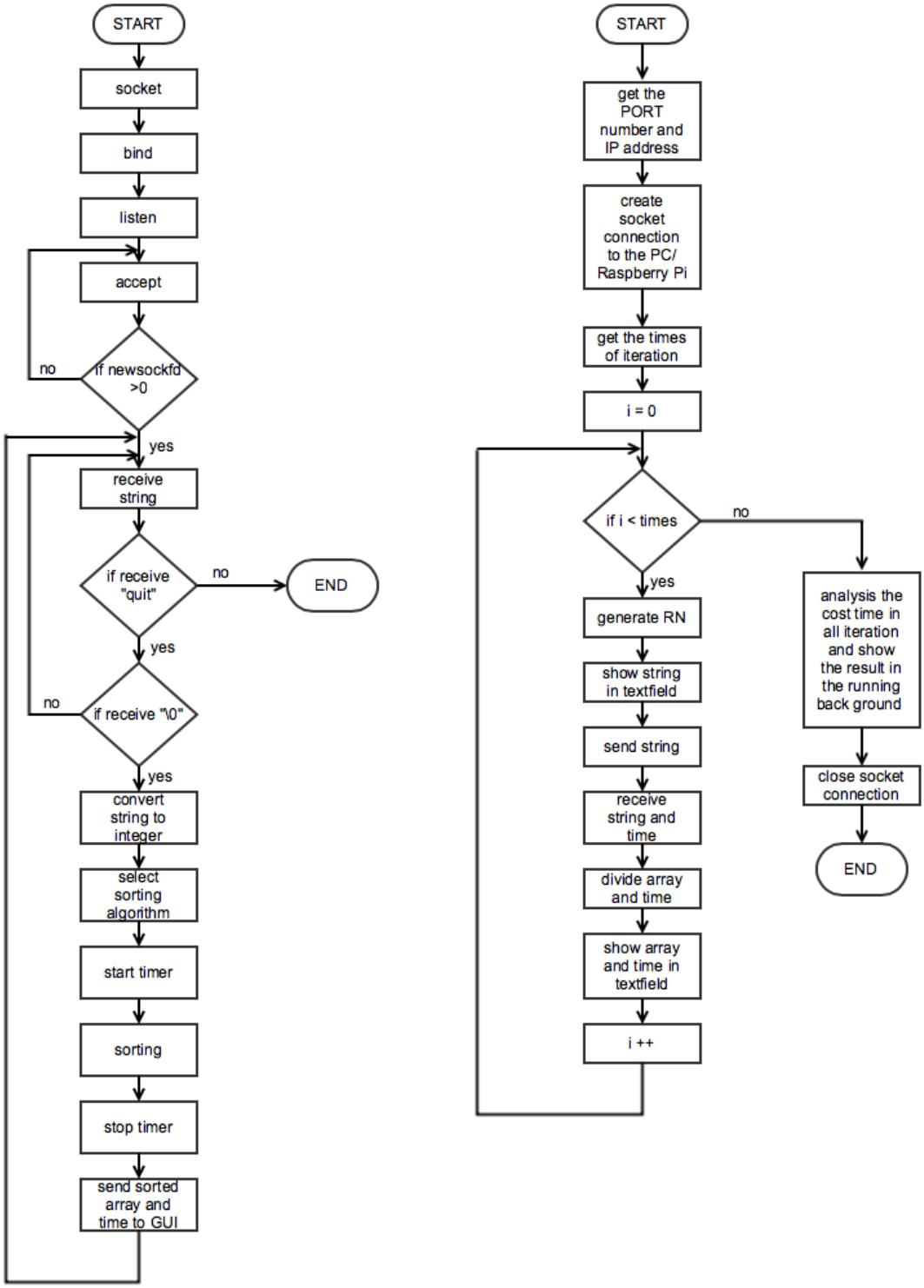

Figure 41. The Procedure of Sorting Algorithms with Socket Communication.

Then the client program on the PC was executed and configured so that a connection to the server could be established in order to do the experiments. Figure 41 illustrates the server (left side) and the client (right side) implementation.

4.4 Time Consumption of Different Sorting Algorithms

Figure 42 shows the Big-O Complexity Chart and Table 5 represents the Time Complexity of sorting algorithms that are used to compare the theoretical results with the experimental results.

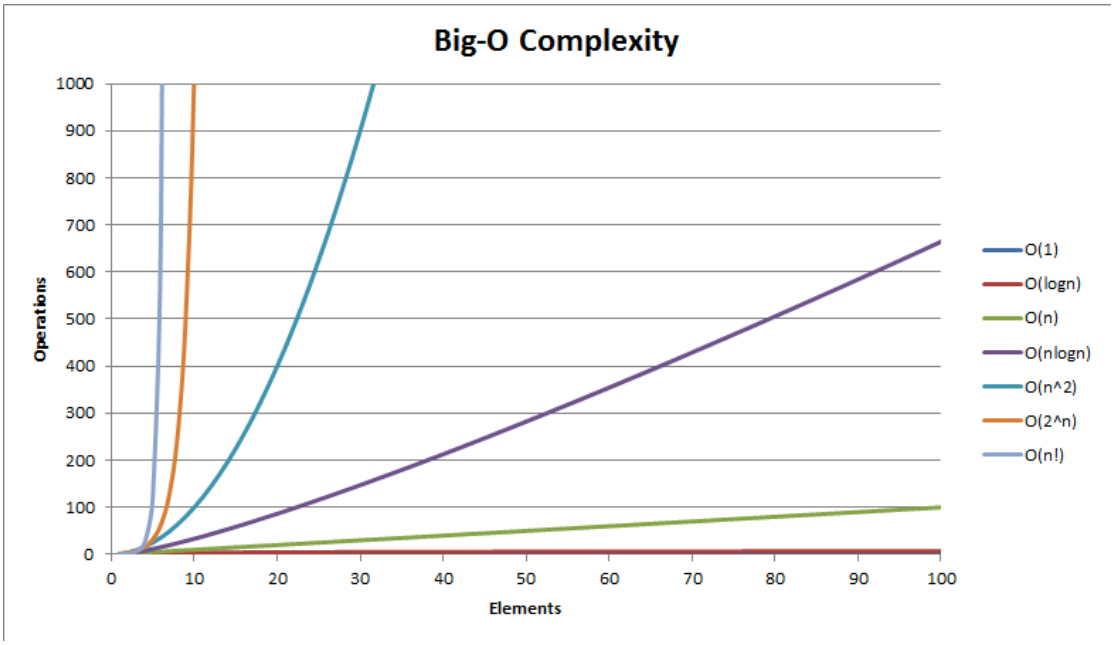

Figure 42. Big-O Complexity Chart (Drowell E.).

The relation between those big-oh values can be expressed in the following way:

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

Table 5 shows the big-oh values for the four introduced sorting algorithms.

Table 5. Time Complexity Table of the four introduced Sorting Algorithms (Drowell E.).

	Best	Average	Worst
Bubble Sort	O(n)	O (n^2)	O (n^2)
Insertion Sort	O(n)	O (n^2)	O (n^2)
Heap Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	O (n^2)

As a result, bubble sort and insertion sort have similar performance, while heap sort and quick sort have better performance.

In the experiments, the amount of random numbers are 500, 1000, 5000, 10000, 20000, 35000 and 50000. Each sorting algortihm has been tested 150 times for each amount of random numbers to ensure accurate results.

4.4.1 Time Consumption of Different Sorting Algorithms on the PC

First, the experiments were done on the PC. Table 6 shows the results.

Table 6. Time Consumption of the four Sorting Algorithms executed on PC.

Bubble sort with PC							
	500	**1000**	**5000**	**10000**	**20000**	**35000**	**50000**
Max	0,002612	0,008666	0,115108	0,46208	1,854659	6,187814	11,638444
Average	0,00209	0,006672	0,105086	0,435587	1,778897	5,67122	11,358998
Min	0,001323	0,003849	0,102905	0,428085	1,763745	5,495457	11,305336

Heap sort with PC							
	500	**1000**	**5000**	**10000**	**20000**	**35000**	**50000**
Max	0,000201	0,000416	0,00186	0,003899	0,007464	0,013294	0,016088
Average	0,000153	0,000322	0,001189	0,002581	0,005651	0,010224	0,013165
Min	0,000065	0,000162	0,000812	0,001736	0,003723	0,006912	0,009174

Quick sort with PC							
	500	**1000**	**5000**	**10000**	**20000**	**35000**	**50000**
Max	0,000158	0,00034	0,001503	0,003455	0,004861	0,009197	0,012735

Average	0,000121	0,000251	0,000795	0,002129	0,004286	0,007326	0,009747
Min	0,00005	0,000132	0,000593	0,001258	0,002632	0,004718	0,006925

Insertion sort with PC							
	500	**1000**	**5000**	**10000**	**20000**	**35000**	**50000**
Max	0,000625	0,002129	0,025933	0,079503	0,291809	0,85358	1,728695
Average	0,000477	0,0014	0,019763	0,069553	0,262994	0,793808	1,60535
Min	0,000214	0,000681	0,015955	0,063187	0,253727	0,777442	1,586268

The relationship between the time consumption and amount of random numbers is non-linear. Figure 43 presents the average time consumption of the four introduced algortihms.

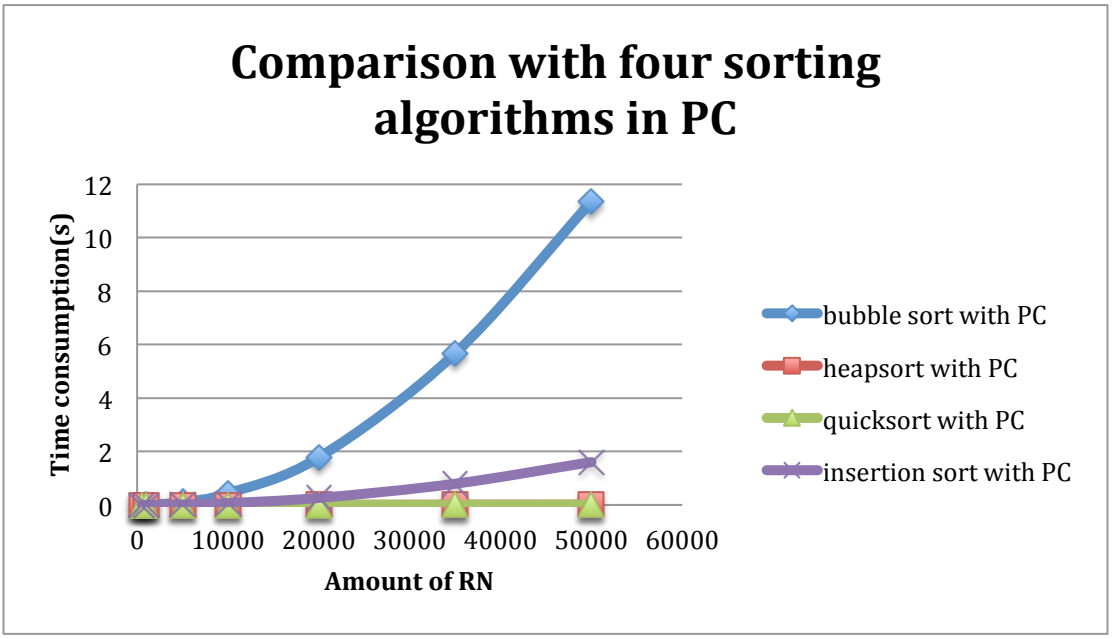

Figure 43. Comparison of the four Sorting Algorithms executed on the PC.

As mentioned before each sorting algorithm has a big-oh complexity. Comparing this big-oh complexity with the results it can be seen that the results agree with the big-oh complexity. Figure 44 and Figure 45 are the zoom-in version of Figure 43.

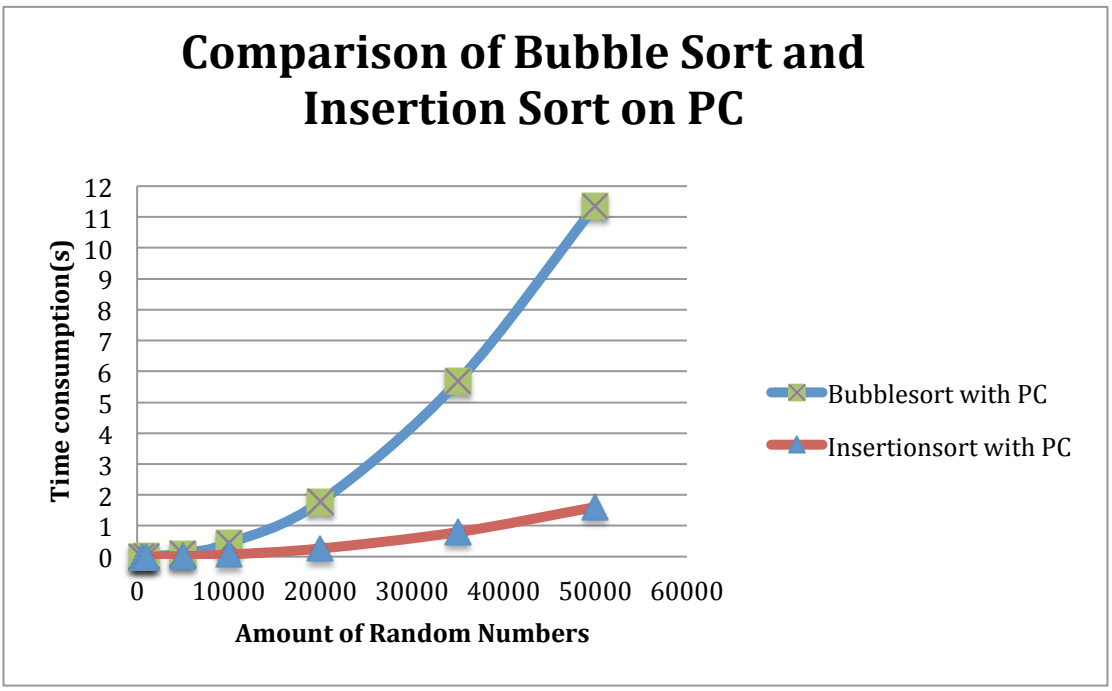

Figure 44. Comparison of Bubble Sort and Insertion Sort on PC.

Even though the performances of bubble sort and insertion behave in a similar way there is still a difference between them. It is clear to see the insertion sort is much faster than the bubble sort.

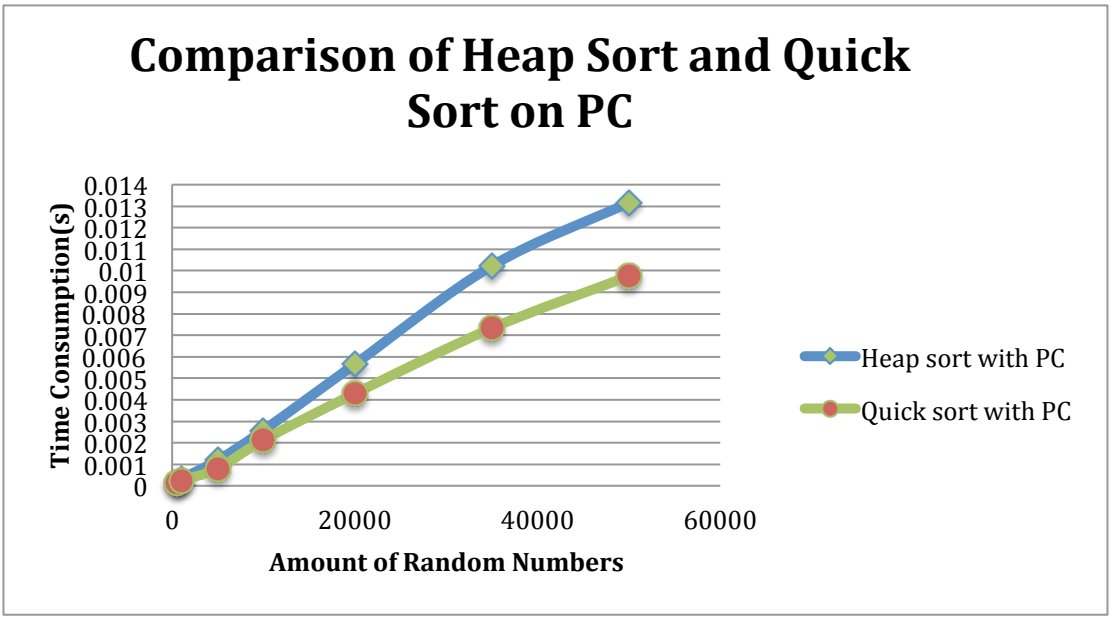

Figure 45. Comparison of Heap Sort and Quick Sort on PC.

Figure 45 illustrates the time consumption of heap sort and quick sort. Comparing the heap sort and quick sort it can be clearly seen that the quick sort is much faster. In comparison with bubble sort, the heapsort is 1000 times faster and the quicksort is even more than 1000 times faster. Even though the quick sort has the O(n^2) complexity in the worst case, the average performance of quick sort is still better than heap sort. As a result, the quick sort performance is the best among those four sorting algorithms.

4.4.2 Time Consumption of Different Sorting Algorithms on Raspberry Pi

The same experiments as described in section 4.4.1 have been done on the Raspberry Pi. Table 7 shows the results.

Table 7. Time Consumption of the four Sorting Algorithms executed on Raspberry Pi.

Bubble sort with Raspberry pi							
	500	1000	5000	10000	20000	35000	50000
Max	0, 024102	0, 082325	2, 046717	8, 634795	35, 449864	118, 550516	251, 038379
Average	0, 020337	0, 078527	2, 030446	8, 535436	34, 733894	117, 622606	250, 008917
Min	0, 019349	0, 076723	2, 012727	8, 503214	34, 634877	117, 312243	249, 495052

Heap sort with Raspberry pi							
	500	1000	5000	10000	20000	35000	50000
Max	0, 001019	0, 003331	0, 018381	0, 03311	0, 097836	0, 172151	0, 22146
Average	0, 000825	0, 001894	0, 013863	0, 027671	0, 058596	0, 114716	0, 210807
Min	0, 000795	0, 001789	0, 011509	0, 025702	0, 057427	0, 111139	0, 168274

Quick sort with Raspberry pi							
	500	1000	5000	10000	20000	35000	50000
Max	0, 000607	0, 002709	0, 018456	0, 023947	0, 050436	0, 108495	0, 186727
Average	0, 000511	0, 001141	0, 009072	0, 018542	0, 038194	0, 082891	0, 157344
Min	0, 000475	0, 001052	0, 006694	0, 015451	0, 037006	0, 07844	0, 133231

Insertion sort with Raspberry pi							
	500	1000	5000	10000	20000	35000	50000

Max	0, 010465	0, 024052	0, 465041	1, 904898	8, 043273	24, 301871	52, 754265
Average	0, 004877	0, 020353	0, 45018	1, 874295	7, 842624	24, 076289	52, 248739
Min	0, 004121	0, 018658	0, 440234	1, 826523	7, 757833	23, 840916	51, 757841

As expected, the only difference between these results and the previous results (on PC) is that the sorting time is much longer. Figure 46 shows the average time consumption. Figure 47 and Figure 50 are the zoom-in version of Figure 46.

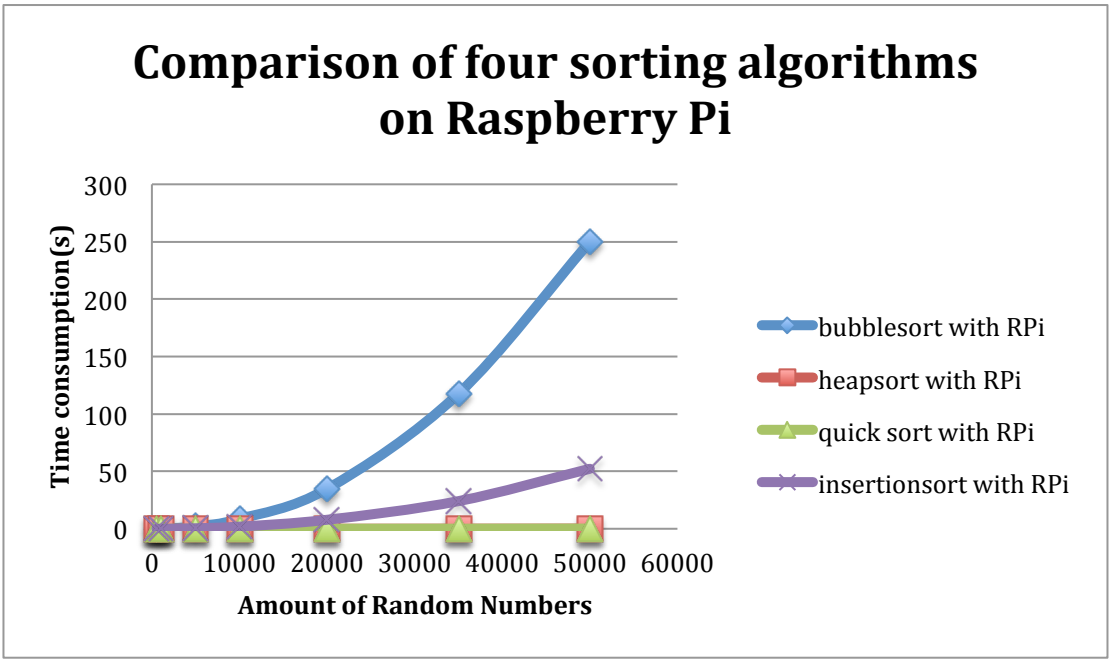

Figure 46. Comparison of the four Sorting Algorithms executed on the Raspberry Pi.

It is easy to find out the growth trend of the curves are similar to the curves' result in section 4.4.1. But the time consumption is much larger than the ones in PC, since the CPU of raspberry Pi is a 700 MHz Low Power ARM1176JZ-F Applications processor, while the CPU of MacBook Pro is a 2.4GHz Intel Core i5 processor.

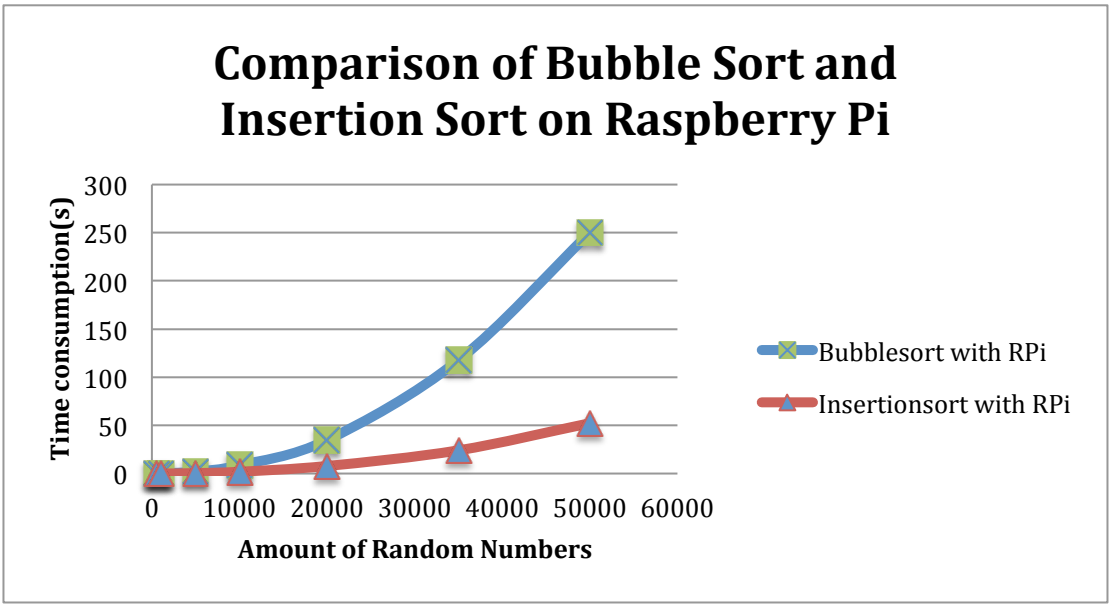

Figure 47. Comparison with Bubble Sort and Insertion Sort executed on Raspberry Pi.

It is not recommended to use the bubble sort and insertion sort algortihms, especially in battery operated devices, because these algortihms are quite inefficient.

Because the Raspberry Pi has a crystal based clock, there is no real time clock running when power is off (Jojopi, 2013). The server code in the Raspberry Pi is using the gettimeofday() and timersub() function for calculating the sorting time of each algorithm. To prove that these functions deliver the right result for the sorting time, an oscilloscope has been used. Before and after executing the sorting algorithms, a selected pin is toggled. Figure 48 and Figure 49 are the screenshots of the time measuring result of bubblesort with 500 random numbers and insertion sort with 5000 random numbers.

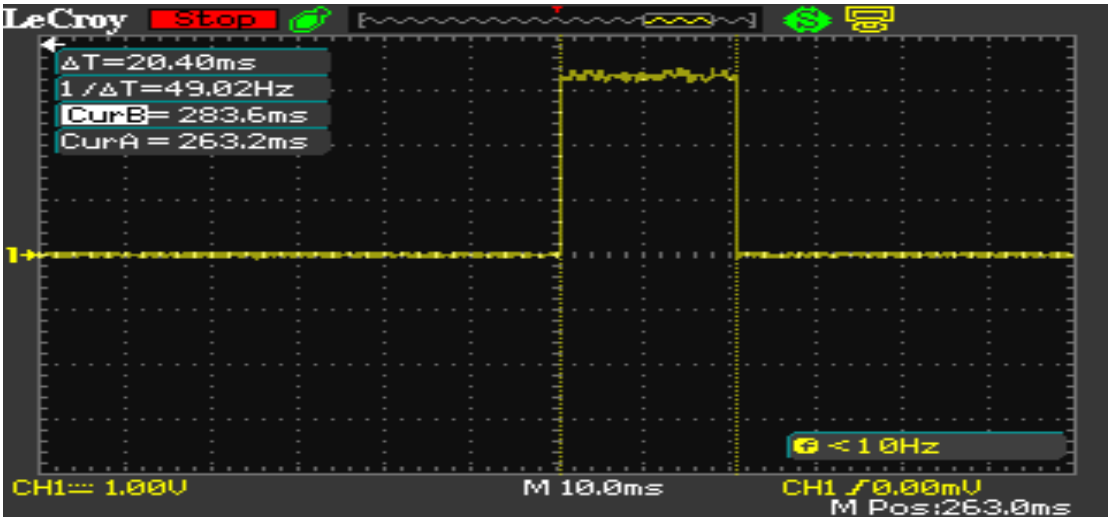

Figure 48. Bubble Sort with 500 Random Numbers.

In Figure 48, ΔT = 0.0204 s which is almost equal to 0.020337s in the Table 7. This proves that the gettimeofday() and timersub() deliver the right results.

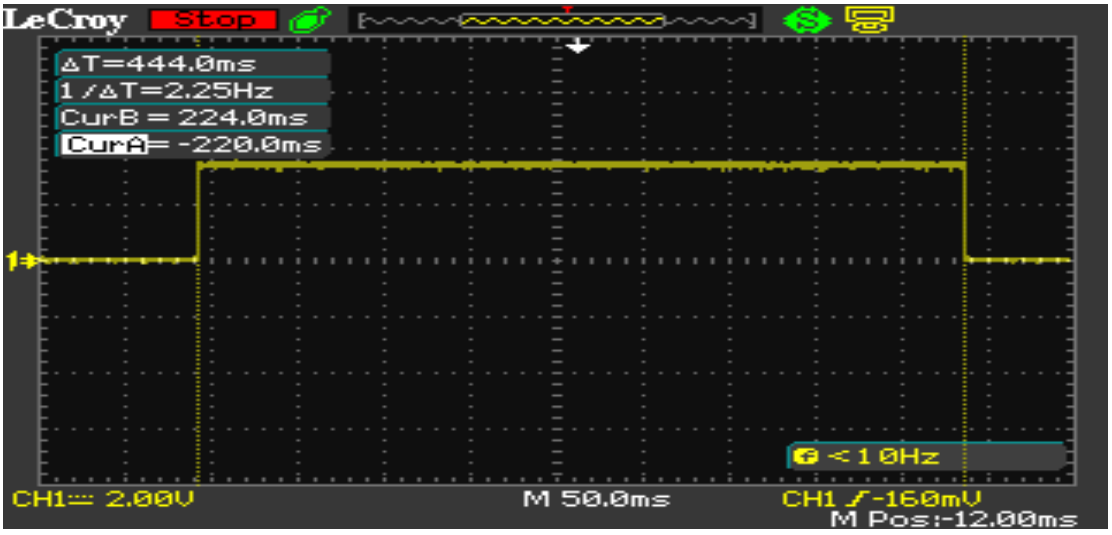

Figure 49. Insertion Sort with 5000 Random Numbers.

In Figure 49, ΔT = 0.444 s which is reasonable range from 0.465041s to 0.440234s (see Table 7).

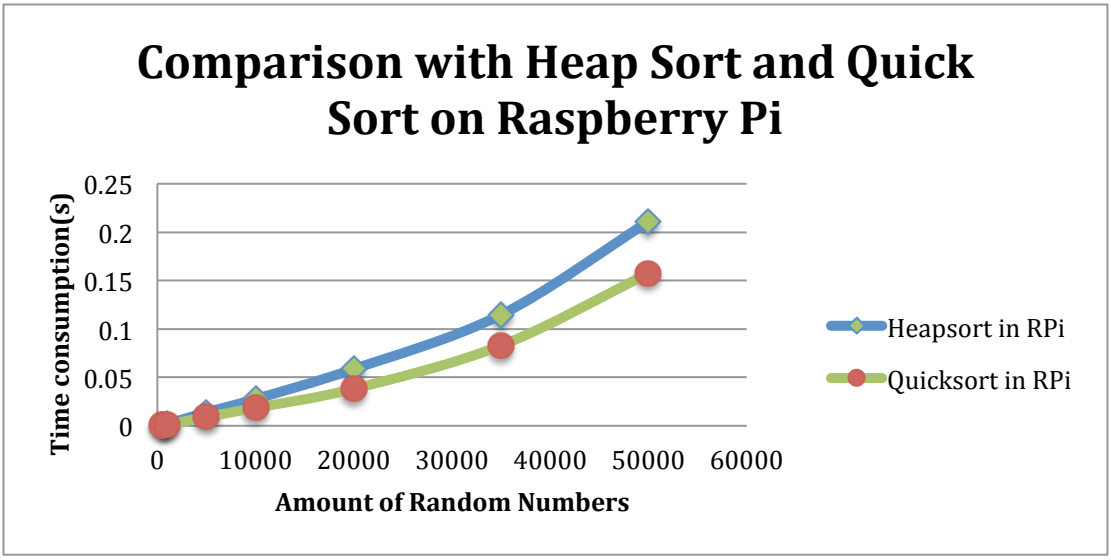

Figure 50. Comparison with Heapsort and Quicksort executed on Raspberry Pi.

In Figure 50, time consumption of heap sort and quick sort are displayed. These curves are similar to the curves presenting the sorting time measurements done on the PC.

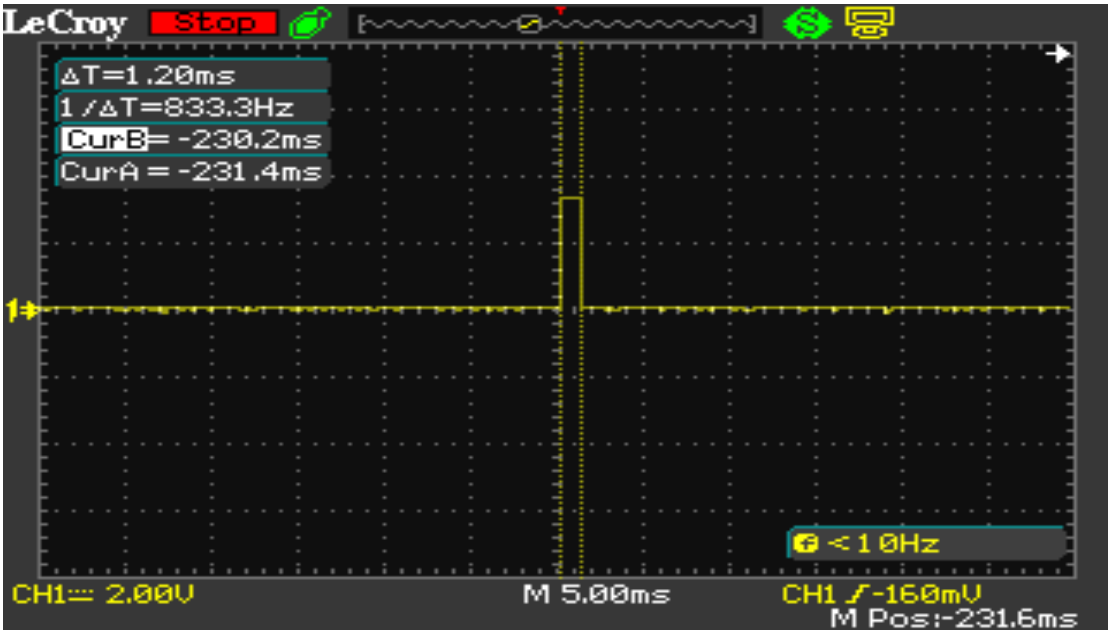

Figure 51. Heap Sort with 500 Random Numbers.

In Figure 51, ΔT = 0.0012s which is closed to the max execution time of heap sort in Table 7.

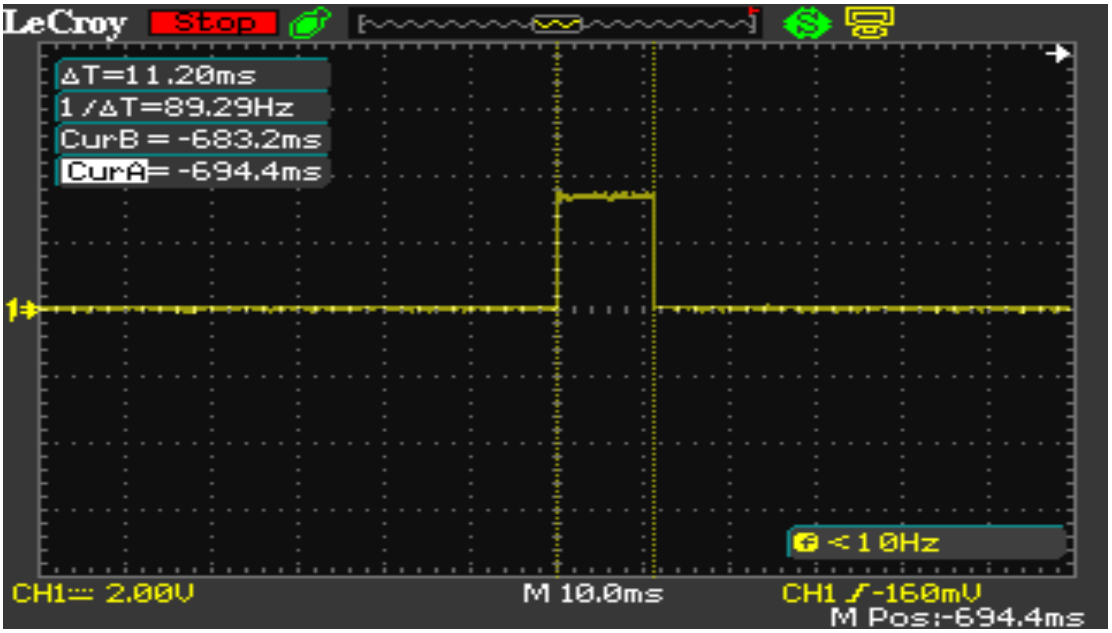

Figure 52. Quick Sort with 5000 Random Numbers.

In Figure 52, ΔT = 0.0112s which is in a reasonable time range (see Table 7).

The result shows that the larger amount of the random number, the higher the time consumption. In comparison to the three other sorting algorithms, the quick sort has the best performance when executed on PC or Raspberry Pi.

5. CONCLUSION AND FUTURE WORK

In this thesis, socket communication, bubble sort, insertion sort, quick sort and heap sort were described and explained. Those four sorting algorithms were analyzed and compared by the time complexity in raspberry Pi and personal computer. In the practical part, a socket server with four sorting algorithms (bubble sort, insertion sort, heap sort and quick sort) has been implemented. Furthermore, a GUI acting as a client has been implemented. This GUI generates random numbers that are sent to the server where they will be sorted before they are sent back and displayed on the GUI. The server side was implemented with the C Programming language and the client side was implemented using the Java Programming language. Sockets have been used for the client to server communication.

Time consumption was measured by the time-measured function provided by C library. The time-measured functions are different, raspberry Pi used a function called gettimeofday() since it has a crystal based clock, while the computer used a normal function called clock(). As a result, the time consumption depends on the amount of numbers as well as on the selected sorting algorithm. The time consumption is higher when it comes to the raspberry Pi because its CPU speed is lower than the CPU speed of the computer. At the end, it is obvious to find out that quick sort has the best performance in both raspberry Pi and computer.

However, besides the time complexity of sorting algorithms, there might be other issues affect the energy consumption of embedded systems. According to the findings of Bunse and his colleagues, energy consumption is not only related to the sorting algorithms themselves but also influenced by memory of embedded system and the data types (Bunse, Hopfner & Mansour 2009). For discussing the energy consumption precisely, more experiments should be planed in the future. For example, measure the current flow of embedded systems with different flash memory sizes. Also, current flow could be measured when different types of data are utilized in the sorting algorithms.

Socket communication is still an interesting field to discover. In the thesis, there is only one server utilized to transmit the data and to sort the arrays. There is also another solution for this, if one client can connect with two or more servers to sort the random numbers. In this situation, the client GUI will connect to several servers and split the data into several groups to let the servers process the data separately. This would lead to a faster sorting time. But there is a complicated part to discuss which is to collect all the data and sort them one more time before the data is sent back to client side. This would depend on the efficiency of the processor. This experiment not only can compare time complexity of different sorting algorithms but also can find out the efficiency of different processors.

References

Brian “Beej Jorgensen” H. (2012). *Beej’s Guide to Network Programming Using Internet Sockets* [online]. Brian “Beej Jorgensen” Hall. Available from the Internet:

<URL:http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html#theory>

Bunse, C.; Hopfner, H., Mansour, E. & Roychoudhury, S. (2009), *Exploring the Energy Consumption of Data Sorting Algorithms in Embedded and Mobile Environments*, Mobile Data Management: Systems, Services and Middleware, 2009. MDM '09. Tenth International Conference on , vol., no., pp.600,607, 18-20 May 2009

Burkepile, A. (2013). *Raspberry Pi Airplay Tutorial*[online]. Razeware LLC. Available from the Internet:

<URL: http://www.raywenderlich.com/44918/raspberry-pi-airplay-tutorial>

Drowell E., *Know Thy Complexities!*. Available from the Internet: <URL: http://bigocheatsheet.com>

Duffymo. (2010). *Why C Is Preferred Instead of Java In Most of The Real Time Appliaciotn*[online]. Stack Exchange, Inc. Available from the Internet: <URL: http://stackoverflow.com/questions/1992286/why-c-is-preferred-instead-of-java-in-most-of-the-real-time-appliaciotn >

Eckhardt, D. (1992). *Linux Programmer’s Manual*[online]. Available from the Internet:<URL: http://man7.org/linux/man-pages/man2/settimeofday.2.html>

Embedded L.W., (2015). RPi Hardware[online]. Elinux.org. Available from the Internet: <URL: http://elinux.org/RPi_Hardware>

Embedded L.W., (2015). RPi Low-level peripherals[online]. Elinux.org. Available from the Internet: <URL: http://elinux.org/RPi_Low-level_peripherals>

Fatourou P. & Kosmas E. (2012). *Introduction to Sockets Programming in C using TCP/IP* [online]. Avalable from the Internet:

<URL:http://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf>

Github. *Raspberrypi/Linux*[online]. GitHub, Inc. Available from the Internet: <URL: https://github.com/raspberrypi/linux>

Haahr, M. *Introduction to Randomness and Random Numbers*[online]. RANDOM.ORG. Available from the Internet: <URL: https://www.random.org/randomness/>

IBM Knowledge Center. *Read-Read Data on a Socket* [online]. IBM Corporation 1994,2015. Available from the Internet: <URL:http://www-01.ibm.com/support/knowledgecenter/#!/SSB23S_1.1.0.10/com.ibm.ztpf-ztpfdf.doc_put.10/gtpc2/cpp_read.html?cp=SSB23S_1.1.0.10%2F0-3-8-1-0-8-17>

IBM Knowledge Center. *Write–Write data on a connected socket* [online]. IBM Corporation 1994,2015. Available from the Internet: <URL:http://www-01.ibm.com/support/knowledgecenter/#!/SSB23S_1.1.0.10/com.ibm.ztpf-ztpfdf.doc_put.10/gtpc2/cpp_write.html?cp=SSB23S_1.1.0.10%2F0-3-8-1-0-17-8>

IEEE Std 1003.1. (2004). *The Open Group Base Specifications Issue 6* [online]. 2001-2004 The IEEE and The Open Group. Available from the Internet: <URL:http://pubs.opengroup.org/onlinepubs/009695399/functions/read.html>

IEEE Std 1003.1. (2004). *The Open Group Base Specifications Issue 6* [online]. 2001-2004 The IEEE and The Open Group. Available from the Internet: <URL: http://pubs.opengroup.org/onlinepubs/009695399/functions/write.html>

IEEE Std 1003.1. (2004) *The Open Group Base Specifications Issue 6* [online]. 2001-2004 The IEEE and The Open Group. Available from the Internet:

<URL:http://pubs.opengroup.org/onlinepubs/009695399/functions/close.html>

Jojopi. (2013). Code Execution Time[online]. Raspberrypi.org. Available from the Internet:

< URL: https://www.raspberrypi.org/forums/viewtopic.php?f=33&t=33657>

Kumar, A., (2012). *Bubble Sort Algorithm*[online]. Available from the Internet: <URL: http://ashwiniec.blogspot.fi/2012/06/bubble-sort.html>

Kumar, A., (2012). *Insertion Sort Algorithm*[online]. Available from the Internet: <URL: http://ashwiniec.blogspot.fi/2012/06/insertion-sort-algorithm.html>

Lecture 14: HeapSort Analysis and Partitioning[online]. Available from the Internet: <URL: http://www.cs.umd.edu/~meesh/351/mount/lectures/lect14-heapsort-analysis-part.pdf>

Moore, R., (1999). *Heapsort*[online]. Sydney: Macquarie University. Available from the Internet: <URL: https://www.cs.umd.edu/class/fall2006/cmsc351/notes/heapsort/ >

Netbeans. *Introduction to GUI Building*[online]. Available from the Internet: <URL: https://netbeans.org/kb/docs/java/gui-functionality.html>

Nprasan. (2014). *Set Up Real Time Clock(RTC) on Raspberry Pi*[online]. Autodesk, Inc. Available from the Internet: <URL:

http://www.instructables.com/id/Set-up-Real-Time-Clock-RTC-on-Raspberry-Pi/>

ORACLE. *What is a Socket?* [online]. 1995-2015 Oracle and/or its affiliates. Available from the Internet: <URL:https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html>

Quicksort Analysis[online]. Available from the Internet: <URL: http://www.cise.ufl.edu/class/cot3100fa07/quicksort_analysis.pdf>

Quicksort Analysis[online]. Available from the Insternet: < URL: http://www.cise.ufl.edu/class/cot3100fa07/quicksort_analysis.pdf>

Raspbian. *Welcome to Raspbian*[online]. Raspbian.org. Available from the Internet: <URL: https://www.raspbian.org>

Sedgewick, R. (1998). *Algorithms in C.* Third Edition. Addison-Wesley Publishing Company, Inc.

Sheldon T. (2001). *TCP (Transmission Control Protocol)* [online]. Pan America and International. Available from the Internet: <URL: http://www.linktionary.com/t/tcp.html>

Stallings W. (2007). *Sockets: A Programmer's Introduction* [online]. Available from the Internet:

<URL:http://www.comp.hkbu.edu.hk/~comp2330/lecture/notes/C-Sockets.pdf>

Tutorialspoint. *C Library Function – Clock()*[online]. Available from the Internet: <URL: http://www.tutorialspoint.com/c_standard_library/c_function_clock.htm>

Tutorial 2: Heaps[online]. Available from the Internet: <URL: http://www.cs.toronto.edu/~krueger/cscB63h/w07/lectures/tut02.txt>

Wikipedia (2015). *Internet Protocol Suite*[online]. Available from the Internet the Internet: <URL:https://en.wikipedia.org/wiki/Internet_protocol_suite>

Wikipedia. (2015a). *CPU Time*[online]. Available from the Internet: <URL: https://en.wikipedia.org/wiki/CPU_time>

Wiring Pi[online]. WorldPress. Available from the Internet: <URL: http://wiringpi.com>

Wikipedia. (2015b). *Secure Shell*[online]. Available from the Internet: <URL: https://en.wikipedia.org/wiki/Secure_Shell>

Wikipedia. (2015c). *Sorting Algorithm*[online]. Available from the Internet: <URL: https://en.wikipedia.org/wiki/Sorting_algorithm>

Weiss, M.A., (1994). *Data Structures and Algorithm Analysis in C++*. California: The Benjamin/Cummings Publishing Company, Inc.

Yu-Hua Wang; Zhi-Dong Shen; Huan-Guo Zhang, "*Pseudo Random Number Generator Based on Hopfield Neural Network*," Machine Learning and Cybernetics, 2006 International Conference on , vol., no., pp.2810,2813, 13-16 Aug. 2006