EE2K1:
Lab Report

Yousef Amar
(1095307)

Contents

Lab 2

Introduction

One of the most important concepts when dealing with operating systems is processor scheduling and scheduling policies. Through using the Torsche Toolbox with MatLab to simulate processor scheduling one can analyse discrete events. The aims of this lab are to

- Differentiate between different types of scheduling policies
- Identify the main parameters involved in real time and non-real time scheduling
- Work out the feasibility study of a possible scheduling policy
- Distinguish various performance criteria employed to evaluate a given scheduling strategy
- Calculate the timing performance of each task
- Illustrate the task execution through graphical representation
- Familiarize with Torsche MatLab toolbox to handle scheduling problems

Full documentation of how to use the Torsche Toolbox can be found at http://rtime.felk.cvut.cz/scheduling-toolbox/manual/. This report will not explain the uses of the functions.

Several process scheduling algorithms will be discussed in this report. These include:

First-Come, First-Served (FCFS)

As the name suggests, the process with the earliest arrival time is executed first. It can commonly display the "Convoy Effect" (short processes behind long processes).

Pre-emptive Deadline Scheduling

Similar to priority scheduling, a process with an earlier deadline can interrupt a running process on arrival regardless of context switch time. It can also be assigned a starting deadline in conjunction with the completion deadline, however in this lab this concept will not be looked at.

In quick comparison, other common scheduling policies include:

Round Robin

Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue. If the time quantum is small with respect to the context switch (the time it takes to switch processes), the overhead becomes too high and it is no longer efficient. If the time quantum is high, it behaves in a First In First Out (FIFO) manner.

Shortest Job First (SJF)

SJF can be either pre-emptive or non-pre-emptive. As the name suggests, the processes with the shortest burst time are executed first. If pre-emptive, the scheme is known as the Shortest Time Remaining First (STRF). SJF is optimal for giving minimum average waiting time for a given set of processes. STRF efficiency can be debated when considering the relationship between burst time and contest switch time.

Specifications, Procedure, Results and Conclusions for Part 1

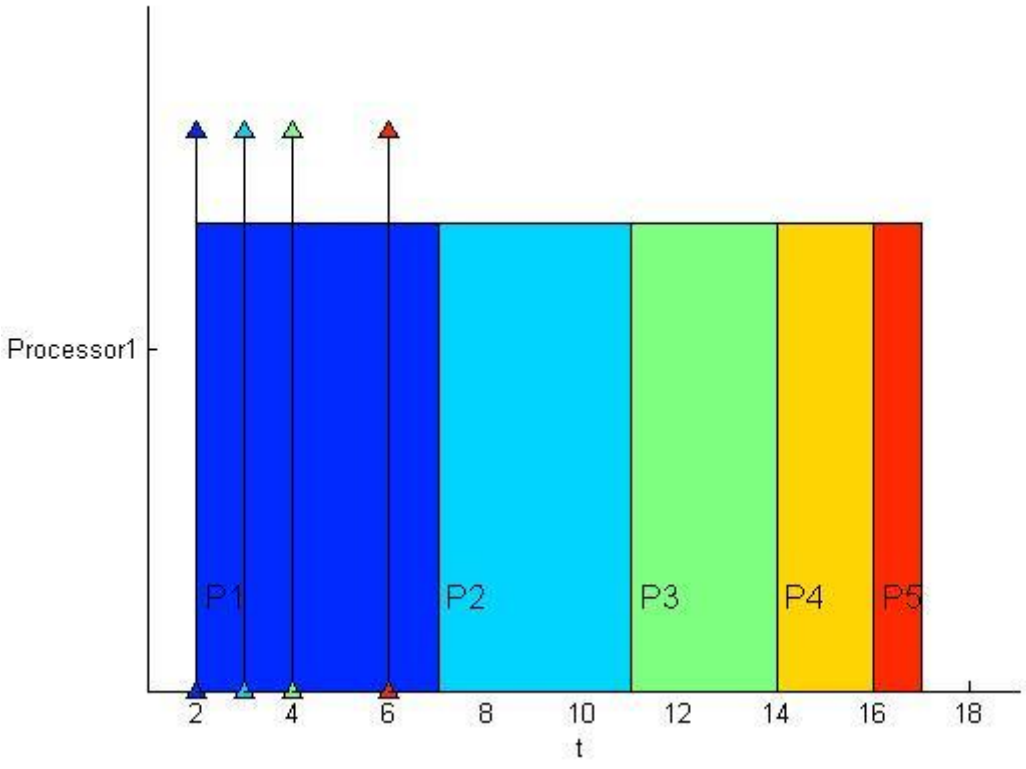
A set of processes is given. The timing diagram of these processes is to be drawn using the *alglrjcmax* command using the FCFS (First Come First Served) algorithm. This is as simple as it gets as the name suggests, it is inherently non-pre-emptive and once a process has started, it executes to completion after which the process with the next arrival time is executed.

Process	Arrival Time	Burst Time
P1	2	5
P2	3	4
P3	4	3
P4	6	2
P5	6	1

L2P1A.m:

```
clear
P1 = task('P1',5,2)
P2 = task('P2',4,3)
P3 = task('P3',3,4)
P4 = task('P4',2,6)
P5 = task('P5',1,6)
p = problem('1|rj|Cmax')
P = taskset([P1 P2 P3 P4
P5])
TS = alg1rjcmax(P,p)
plot(TS)
```

L2P1A.jpg:



The average waiting time in this scenario is **(0+(7-3)+(11-4)+(14-6)+(16-6))/5 = 5.8**
The average turnaround time is **((7-2)+(11-3)+(14-4)+(16-6)+(17-6)/5 = 8.8**
The average response is **(0+(7-3)+(11-4)+(14-6)+(16-6))/5 = 5.8**

Clearly this algorithm is not optimal for minimizing waiting time. Intuitively, one can see that the process with the greatest burst time runs first, therefore making the ones with shorter burst times wait until it is completed before they can start. Had they went first, only that one large process would have had to wait. Instead, the shorter processes have to all wait for one long one. This is even more significant due to the fact that they all arrived while the first process was executing.

We now assume that these processes have associated deadlines as seen below. This time however we use a pre-emptive algorithm that is implemented through the *horn* algorithm. The fact that this algorithm is pre-emptive means that, on arrival, a process can interrupt an already running process if it has an earlier deadline before the previously running process can resume.

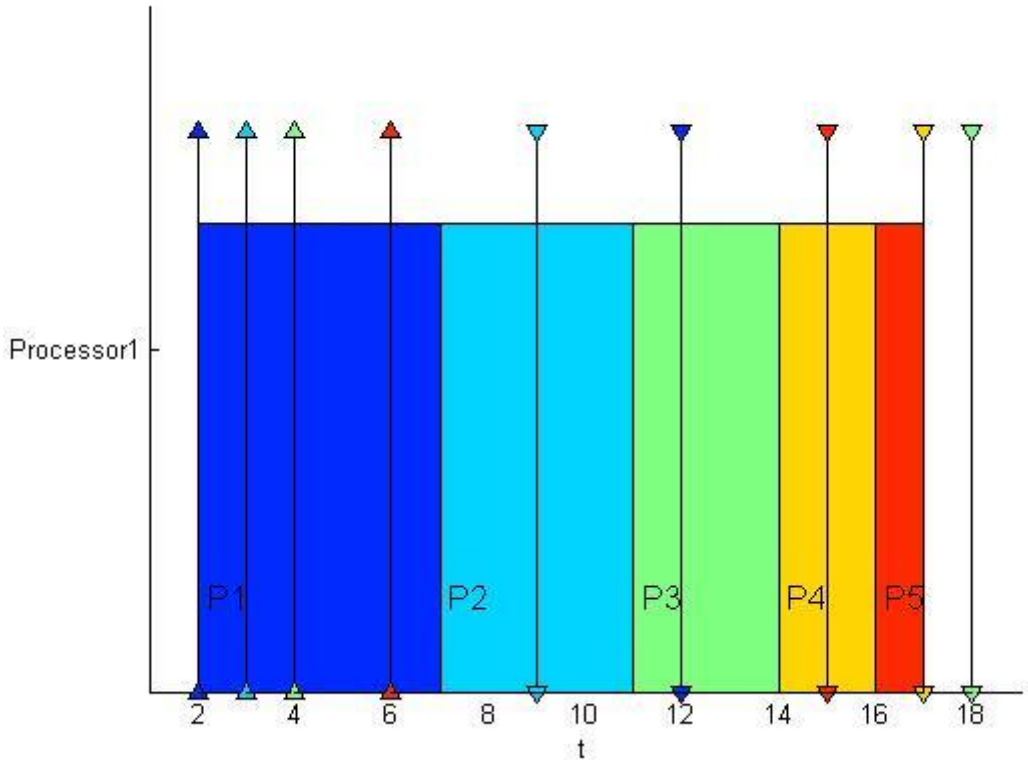
Process	Arrival Time	Burst Time	Deadline
P1	2	5	12
P2	3	4	9
P3	4	3	18
P4	6	2	17
P5	6	1	15

A quick clarification: within the Torsche Toolbox documentation, the deadline is equivalent to the "due date" and the "deadline" in the toolbox does not empower a process to pre-empt. The following is the result of setting the "deadline":

L2P1B.m:

```
clear
P1 = task('P1',5,2,12)
P2 = task('P2',4,3,9)
P3 = task('P3',3,4,18)
P4 = task('P4',2,6,17)
P5 = task('P5',1,6,15)
p = problem('1|pmtn,rj|Lmax')
P = taskset([P1 P2 P3 P4 P5])
TS = horn(P,p)
plot(TS)
```

L2P1B.jpg:

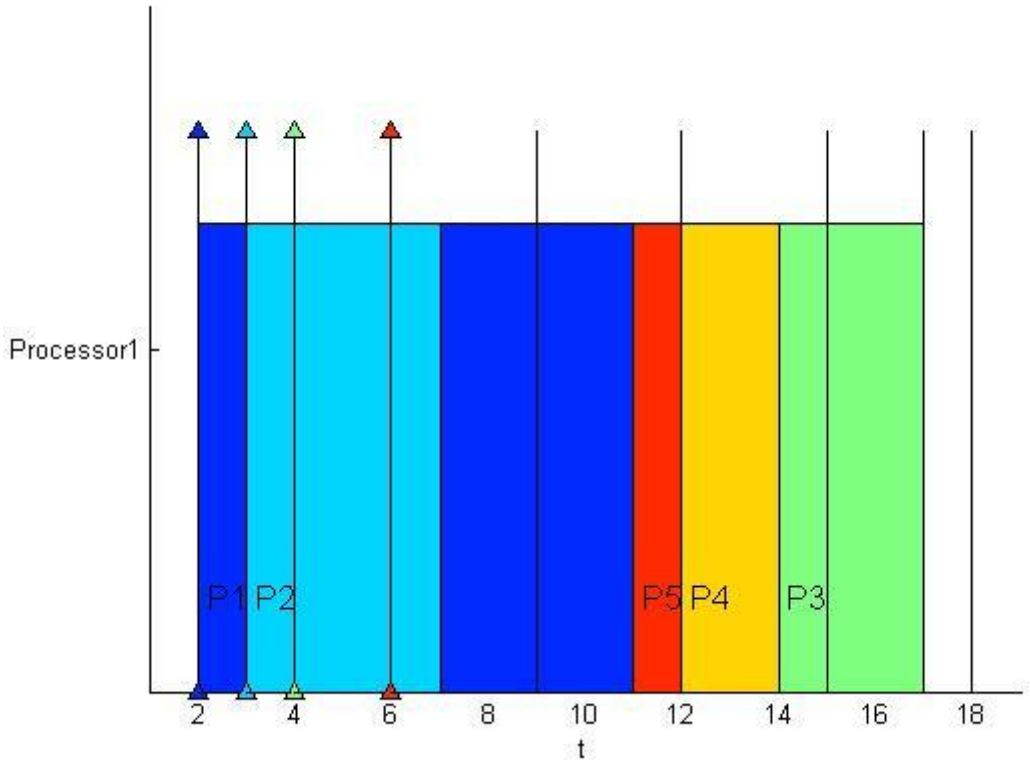


Evidently it is no different than had all the “deadlines” not existed or been any other arbitrary number. In order to apply this algorithm correctly, one has to set the “deadlines” to infinite (or anything else really) and the “due dates” to the desired deadline.

L2P1C.m:

```
clear
P1 = task('P1',5,2,inf,12)
P2 = task('P2',4,3,inf,9)
P3 = task('P3',3,4,inf,18)
P4 = task('P4',2,6,inf,17)
P5 = task('P5',1,6,inf,15)
p = problem('1|pmtn,rj|Lmax')
P = taskset([P1 P2 P3 P4 P5])
TS = horn(P,p)
plot(TS)
```

L2P1C.jpg:



Now, the algorithm recognises that P2 has an earlier deadline than P1 and allows it to pre-empt on arrival. Similarly, P5 executes before P4 which in turn executes before P3 as they all have lower deadlines (15, 17 and 18 respectively). They wait for P1 to finish execution instead of interrupting it though, since P1 has an even earlier deadline (12).

The average waiting time in this scenario is **((0+7-3)+0+(14-4)+(12-6)+(11-6))/5 = 5**
The average turnaround time is **((11-2)+(7-3)+(17-4)+(14-6)+(12-6)/5 = 8**
The average response time is **(0+0+(14-4)+(12-6)+(11-6))/5 = 4.2**

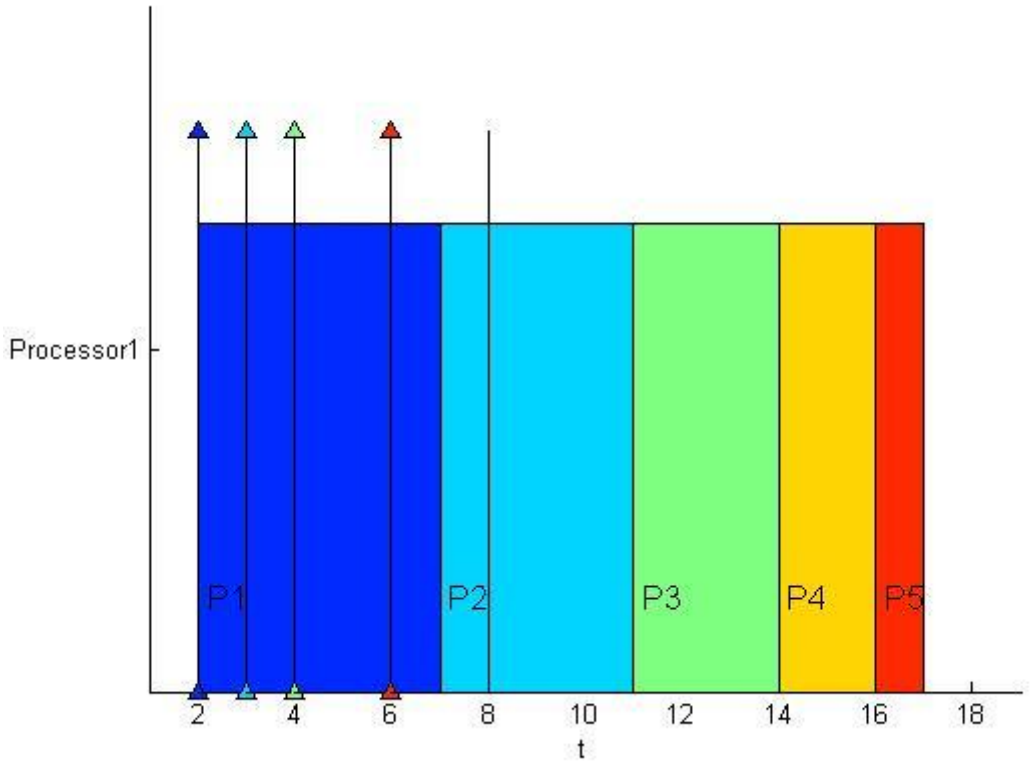
This reinforces the points made earlier and in this case the waiting time is less. Although the waiting time in this case only correlates indirectly with the burst time, the algorithm can be considered “smarter” as the processes that have an earlier deadline take priority. They also have a slightly faster turnaround time. Clearly this is more efficient no matter what way one looks at it.

The same algorithm is repeated with all deadlines set equal at a random value (8 in our case).

L2P1D.m:

```
clear
P1 = task('P1',5,2,inf,8)
P2 = task('P2',4,3,inf,8)
P3 = task('P3',3,4,inf,8)
P4 = task('P4',2,6,inf,8)
P5 = task('P5',1,6,inf,8)
p = problem('1|pmtn,rj|Lmax')
P = taskset([P1 P2 P3 P4 P5])
TS = horn(P,p)
plot(TS)
```

L2P1D.jpg:



The order in which the processes execute reverts back to how it was initially. As the processes have the same deadline, none of them have priority over any other anymore and thus simply execute in the order they arrive. The average waiting and turnaround time is the same as in the first trial.

This algorithm is optimal for processes in which the deadline is a deciding factor. Although sometimes deadlines cannot be kept, like above, the algorithm aims to enable as many deadlines as possible to be kept. Algorithms of this kind have the advantage that they guarantee that no single task blocks the processor for a long time. A risk however is the starvation of processes with very distant deadlines (hence priorities) when there is a large concentration of high priority processes (lockout). The solution to this is “aging”, where processes’ priority increase over time. This risk is non-existent with FCFS algorithms.

Specifications, Procedure, Results and Conclusions for Part 2

The problem presented is that work in a garage needs to be organised in a way that all clients are satisfied with repairs done on their cars on time. The information given is:

- Car 1 needs a battery replacement, which has to be finished at 12:00.
- Car 2 is missing an electrical cable and has to be ready at 15:00.
- Car 3 has dammed petrol pump and has to be fixed till 16:00.
- Car 4 has a damaged seal ring and needs to be fixed before 18:00.
- Batteries will be delivered to the garage at 8:00.
- The seal ring will be delivered at 10:00.
- The petrol pump will also be delivered at 10:00.
- The electrical cable is already in the garage store.
- Battery replacement is expected to take 30 minutes.
- Replacing the pump is expected to take 2 hours.
- Replacing the electrical cable is expected to take 1 hour.
- Replacing the seal ring is expected to take 20 minutes.

In other words:

Process	Arrival Time	Burst Time	Deadline
Car 1	8	0.5	12
Car 2	0	1	15
Car 3	10	2	16
Car 4	10	0.33	18

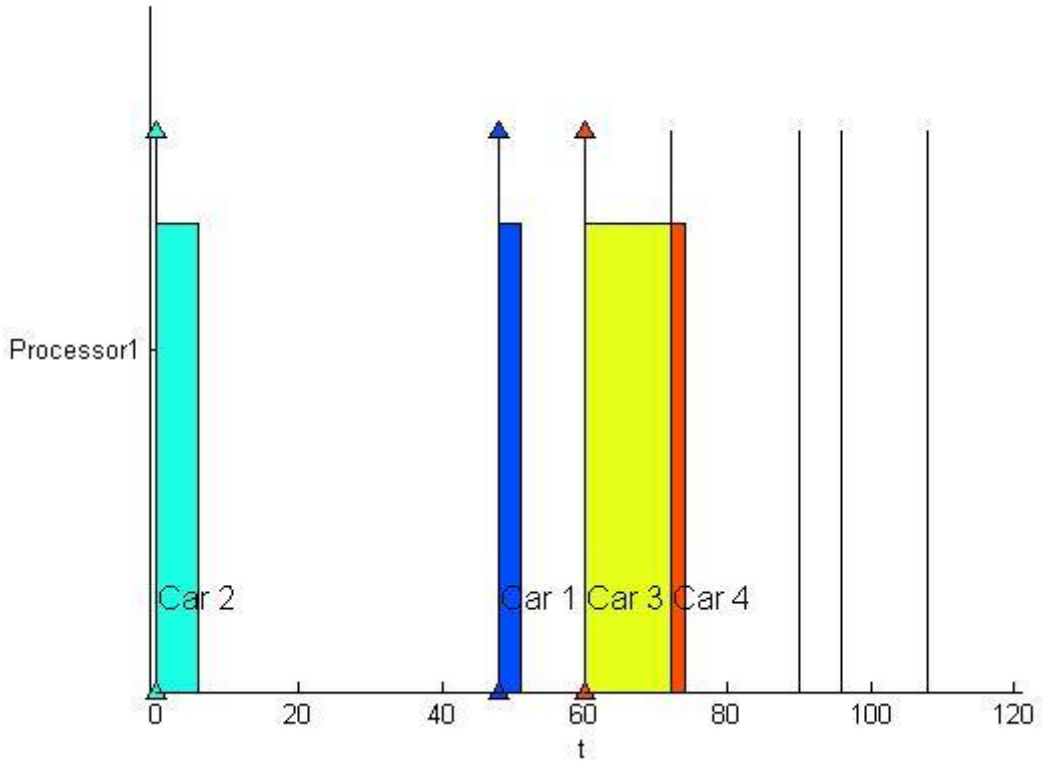
In order to be able to work with integers, the same values can be scaled up 6 times and still enable us to obtain accurate results. The smallest unit of time will just have to ultimately be divided by 6.

Process	Arrival Time	Burst Time	Deadline
Car 1	48	3	72
Car 2	0	6	90
Car 3	60	12	96
Car 4	60	2	108

L2P2A.m:

```
clear
P1 = task('Car 1',3,48,inf,72)
P2 = task('Car 2',6,0,inf,90)
P3 = task('Car 3',12,60,inf,96)
P4 = task('Car 4',2,60,inf,108)
p = problem('1|pmtn,rj|Lmax')
P = taskset([P1 P2 P3 P4])
TS = horn(P,p)
plot(TS)
```

L2P2A.jpg:



NB: "electrical cable is already in the garage store" is assumed to mean that work on Car 2 (the one that needs the electrical cable) can start at 00:00.

As is evident, the horn algorithm is more than adequate to make all customers happy. The arrival times and deadlines are spread out enough and the burst times short enough for there to be no need for a process to pre-empt any other one.

Car 2 can be worked on from the start before any other car can be possibly worked on. The process is complete before the arrival of the parts for car 1. Similarly, Car 1 has enough time to complete. After that, the parts for Car 3 and Car 4 parts arrive at the same time. Since their deadlines are still far away though, the order they are tackled is irrelevant, yet Car 3 goes first because it has an earlier deadline. It is clear that the application of concepts brought forth by process scheduling can be very beneficial in a working environment such as a garage or the like.

To conclude, MatLab can truly be considered a powerful tool for simulating process scheduling and exploring different scheduling policies. Furthermore the most basic functionalities of MatLab can allow us to calculate the waiting ant turnaround times for task automatically. Its advantages are only reiterated by the fact that everything we have looked at can be used in real-life applications beyond processor scheduling.

Lab 3: Section A

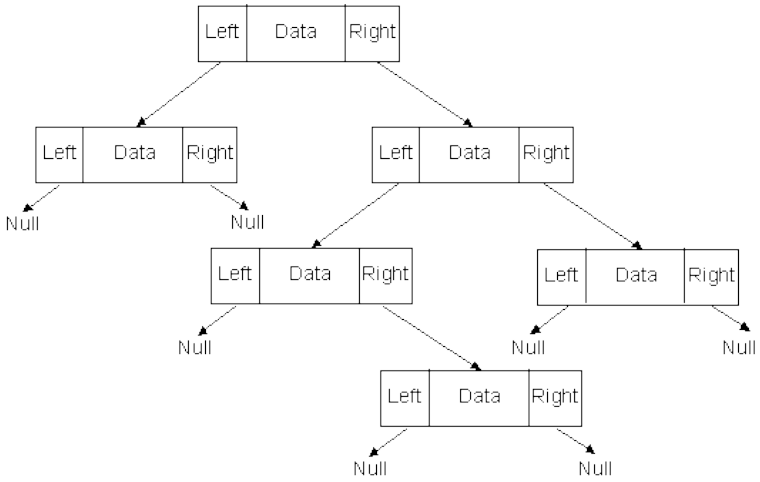
Introduction

A binary tree is one of the most prevalent types of tree data structures. It starts at the root and every subsequent vertex has a maximum of 2 child nodes designated either left or right. As such a binary tree is ordered and the relative position of the child nodes is significant.

The maximum size of a binary tree (the number of nodes) and its height (the number of levels) is therefore easy to infer. Knowing a binary tree nod can have a maximum of two children:

$$\log_2 n \leq h \quad \text{or} \quad n \leq 2^h$$

where n is the tree size and h is the tree height. It is simple to implement in code as it is mostly similar to the linked list data structure only instead there are two pointers in each node storing the address of the left and right children.



Source: Glennrowe.net

All nodes are numbered from left to right going down irrelevant of weather a node is missing or not in a way such that left of a child of i is at 2i+1 and right is 2i+2. This makes it particularly useful for binary search algorithms and storage of a large amount of data that fits the model. As the child pointers are one way only, all nodes can only be accessed by starting from the root. This characteristic is the core trait that causes binary trees to behave the way they do in terms of time complexity and search speed.

The aims of this lab are to create and implement binary trees, identify problems requiring the use of trees and the quantification of the complexity of the binary tree, hence concluding on the importance of tree traversal.

Code Design for Section 1

For the final complete code listing, see attached C files.

Adding a node to the binary tree (iterative approach in notes) - Pseudo code:

```
While (not done)
    If (element's key < root's key)
        If (no left child of root)
            Put new element left of root
            Done!
        Else
            Tree = left sub-tree
    Else if (element's key > root's key)
        If (no right child of root)
            Put new element right of root
            Done!
        Else
            Tree = right sub-tree
    Else
        Do whatever you do when key is already in tree
        Done!
```

Adding a node to the binary tree (my own recursive implementation) - Pseudo code:

```
Start at root as parent
If the key of the node to be added is less than the parent's key
    If the parent has no left child
        Set parent's left child as node to be added
    Else
        Call the same function recursively, setting the left child as the parent
Else if the key of the node to be added is greater than the parent's key
    If the parent has no right child
        Set parent's right child as node to be added
    Else
        Call the same function recursively, setting the right child as the parent
Else
    Flag an error message that the node has been already taken
```

Adding a node to the binary tree – C implementation:

```
void addNode(struct node* parent, struct node* element) {
      if(element->key<parent->key) {
              if(parent->leftChild==NULL)
                      parent->leftChild = element;
              else
                      addNode(parent->leftChild, element);
      } else if(element->key>parent->key) {
              if(parent->rightChild==NULL)
                      parent->rightChild = element;
              else
                      addNode(parent->rightChild, element);
      } /*else
              printf("That key is already taken.\n"); //This affects the execution time.*/
}
```

Note however that a binary tree could allow duplicate keys. In that case the node could go either left or right, but not both and we assume the tree contains at least one node.

Code Design for Section 2

For the final complete code listing, see attached C files.

Generating a binary tree using random numbers - Pseudo code:

Ask the user for a tree size
Set the random number generator seed as the system time
Create a node with a random key (from the rand function) and set it as the root
For the given size
    Create a node with a random key and add it to the tree (using above function)

Creating a node - Pseudo code:

Allocate enough memory to a pointer to a node
Set its key to the number given as an argument
Initialise its child pointers to null
Return the pointer to said node

Printing the tree using in-order traversal (recursive) - Pseudo code:

If left child exists
    In-order traverse left child
Print node key
If right child exists
    In-order traverse right child

This algorithm is used as it makes most sense for printing the tree in this program. If the tree were to be printed level-wise, stacks would have to be used and would have a limit (see discussion in detail later).

Freeing memory allocated to the tree using post-order traversal (recursive) - Pseudo code:

If left child exists
    Post-order traverse left child
If right child exists
    Post-order traverse right child
Set floating pointers to NULL
Free memory allocated to the node itself

Generating a binary tree using random numbers – C implementation:

```
struct node* createNode(int key) {
        struct node* newNode = (struct node*)malloc(sizeof(struct node));
        newNode->key = key;
        newNode->leftChild = NULL;
        newNode->rightChild = NULL;
        return newNode;
}

void printInOrder(struct node* root) {
        if (root->leftChild)
                printInorder(root->leftChild);

        printf("%d ",root->key);

        if (root->rightChild)
                printInorder(root->rightChild);
}

void freeMemoryPostOrder(struct node* root) {
        if (root->leftChild)
                freeMemoryPostOrder(root->leftChild);
        if (root->rightChild)
                freeMemoryPostOrder(root->rightChild);

        root->leftChild = NULL;
        root->rightChild = NULL;
        free(root);
}

int main(void) {
        struct node* root;
        int i, j, size;

        while(1) {
                printf("Enter a tree size (-1 to exit): ");
                scanf("%d", &size);

                if(size<0)
                        break;

                srand(time(NULL));
                root = createNode(rand());
                for(i=0;i<size;i++)
                        addNode(root, createNode(rand()));

                printInorder(root);
                freeMemoryPostOrder(root);
        }
        return 0;
}
```

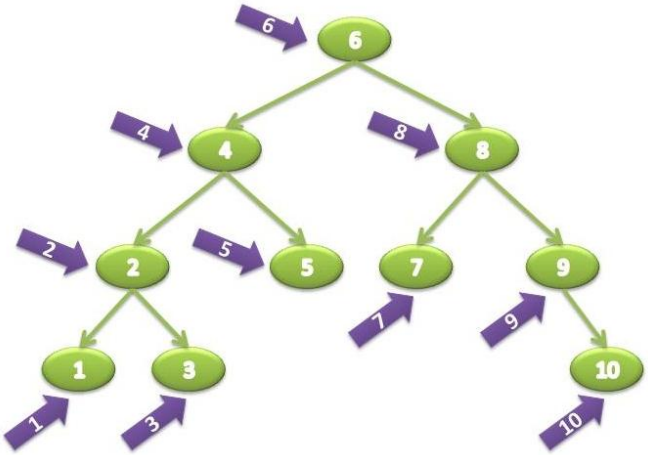
Using the earlier “addNode” function, a binary tree is thus generated. A while loop is used to generate multiple trees to save the user the hassle of closing and reopening the program. Furthermore, for consistency, the memory allocated to a tree is ultimately freed in a post-order fashion.

Results for Sections 1 and 2

As expected, a binary tree with random keys is generated. Example output:

```
Enter a tree size (-1 to exit): 100

423 766 1426 1503 1648 2076 2369 2781 2925 2987 3562 3672 3683 5987 6215 6258 63
26 6396 7033 7071 7675 7803 7969 8644 9003 9029 9543 9702 10010 10343 10907 1109
3 11240 11287 12073 12462 12652 13361 13587 13661 13796 14119 14255 15060 15349
15681 16096 17397 17455 17531 17534 17617 17652 17960 19122 19242 19250 20528 21
278 21483 21720 21837 21839 21862 22220 22944 23622 23636 24358 24484 24734 2508
3 25418 25467 25788 25794 26155 26252 26305 27004 27671 27675 27791 27831 28064
28457 28596 29231 29494 29615 29913 30123 30301 30657 30934 31126 31235 31700 32
411 32593 32727
Enter a tree size (-1 to exit): -1
Press any key to continue . . .
```

It may not seem clear where the levels are at first glance but serves enough for our purpose. The numbers grow gradually such that we start with the bottommost leftmost node, which is by definition the smallest, and ends with the bottommost rightmost node which is in contrast the node with the biggest key. In-order traversal is a recursive algorithm that visits the leftmost child node first before printing anything and only when there is no more left node does it print the node it's on and proceed to the right child only to do the same thing again. As such, the smallest numbers are printed first.



Source: encrypt3d.wordpress.com

This is different than preforming post-order traversal and pre-order traversal as post-order traversal only visits a node when it has traversed left *and* right (such that in the above diagram node 1 would still be the first to be visited, yet 3 would become the second, 2 the third and so on) while pre-order traversal first visits and then traverses left and right (in that case the above tree would be visited in the order 6,4,2,1,3,8,7,9,10). For printing this program they would therefore be impractical. For freeing memory in the "freeMemoryPostOrder" function however, post-order traversal is clearly the method to use as children have to be freed and their pointers set to null before a parent is freed. It is therefore a question of context in order to decide which algorithm to use.

Code Design for Section 3

For the final complete code listing, see attached C files.

Through importing the header file "timer.h" one can calculate the time an algorithm takes to complete. This is done by calling the functions "timer_start()" exactly before the code of which the execution time should be measured (in this case the generation of the binary tree) and "timer_stop()" after it. It is important to remove and printf calls as they would skew the results. To then get the time taken in microseconds, the function "timer_microseconds()" should be called. It returns the time taken and the value can then be printed to the console or used for further calculation. The timer is not biased by anything and measures only the execution time of the desired segment of code as it is only started and stopped on demand.

Timing a binary tree generation algorithm – C implementation:

```
int main(void) {
	struct node* root;
	int i, j, size;
	double totalTime;

	while(1) {
		printf("Enter a tree size (-1 to exit): ");
		scanf("%d", &size);

		if(size<0)
			break;

		totalTime = 0;
		for(j=0;j<100000;j++) {
			srand(time(NULL));
			root = createNode(rand());

			timer_start();
			for(i=0;i<size;i++)
				addNode(root, createNode(rand()));
			timer_stop();
			totalTime += timer_microseconds();

			//printInorder(root);
			freeMemoryPostOrder(root);
		}
		printf("Creating a binary tree of size %d took %fus on average.\n\n", size, totalTime/100000);
	}
	return 0;
}
```

It is self-evident that here we are measuring the construction of the binary tree in its entirety. The traversal of the tree starts at the root and moves either left or right down the levels on every iteration until the key's relative position is found and the node can be inserted. It is to be expected that as the number of elements increases, the time it takes to insert a new node increases as well, but tending towards a log curve, as every time you move down to the next level, you are (on average) ignoring half of the tree and are sequentially halving halves, which would resemble a log curve.

Furthermore, 100 000 trees are created for every size (RAM usage is too high for any more), their generation time measured, and an average found. This takes a long time for large sizes but gives very accurate results. The memory allocated to earlier trees is freed accordingly once the data is extracted of course.

Final Results, Interpretation and Conclusions

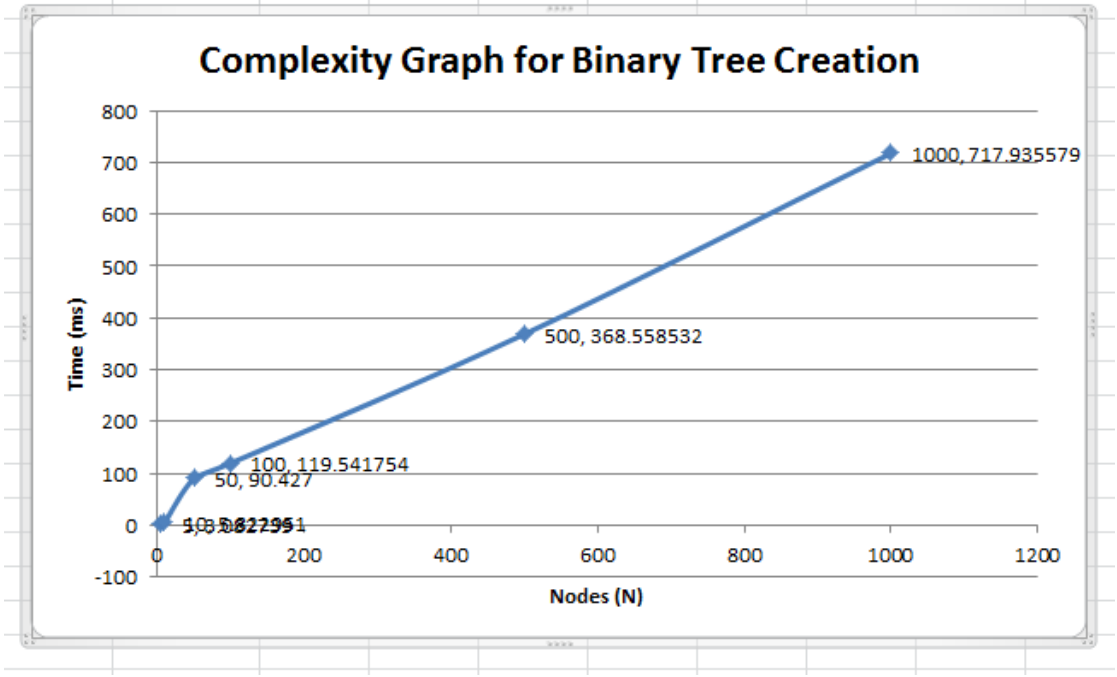
Timing a binary tree generation algorithm – Example output:

```
Enter a tree size (-1 to exit): 5
Creating a binary tree of size 5 took 3.082739us.

Enter a tree size (-1 to exit): 10
Creating a binary tree of size 10 took 5.822951us.

Enter a tree size (-1 to exit): 50
Creating a binary tree of size 50 took 90.427000us.

Enter a tree size (-1 to exit): 100
Creating a binary tree of size 100 took 119.541800us.

Enter a tree size (-1 to exit): 500
Creating a binary tree of size 500 took 368.558500us.

Enter a tree size (-1 to exit): 1000
Creating a binary tree of size 1000 took 717.935600us.

Enter a tree size (-1 to exit): 5000
Creating a binary tree of size 5000 took 3864.727000us.

Enter a tree size (-1 to exit): -1
Press any key to continue . . .
```

Note however that the numbers inherently have no meaning; it is the way the numbers change as the size increases that makes a difference. The same program was run at one instance on the university computers yielding results and on another instance on my personal netbook which is not nearly as fast. The gradient of the resulting of the resulting curve was factors greater than the initial one. The trend however, or the second derivatives, of said curves were identical.

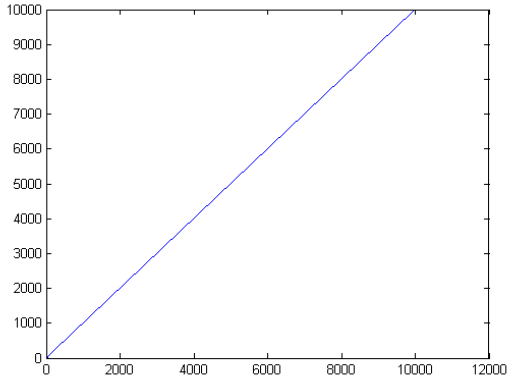
As specified, a graph is created showing the execution time of the generation of a binary tree for sizes n=5, 10, 50, 100, 500, 1000 and 5000. A series of values is noted for each size and the averages plotted. (NB: Time is in microseconds, not milliseconds. The full excel file has also been submitted.)

	N	5	10	50	100	500	1000	5000
approx.->	Time (ms)	3.082739	5.822951	90.427	119.5418	368.5585	717.9356	3864.727

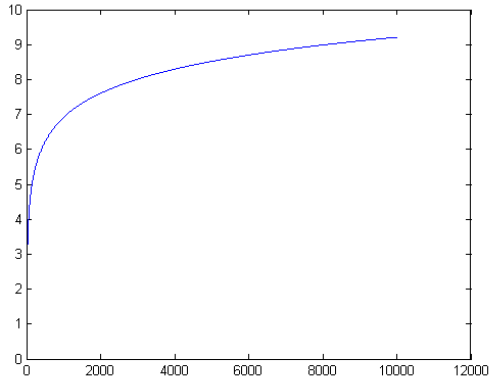


You may notice however, that the predicted complexity is supposed to be O(log n) with worst case O(n) while the graph looks like something in-between as it acts like log n initially before going linear. One may deduce that after a certain amount of random numbers, the random number generator gets existing numbers at a higher frequency as the range, or interval, from which a random number is taken, is limited. Hence, less new keys were left to add effectively stopping the traversal somewhere along the way as duplicate occurrences were found more and more, causing the graph to go linear faster. Therefore, one can conclude our earlier theoretical predictions were accurate and that the following algorithm is especially efficient for large amounts of data as compared to small amounts of data. In theory, the graph should range between the following two graphs:

N



logN



Lab 3: Section B

Introduction

In practical applications, it is often very useful to computationally sort data of any size and deciding on a suitable algorithm is of utmost importance. Examples of common sorting algorithms include:

- Bubble Sort (discussed later)
- Quicksort (discussed later)
- Selection sort (useful where swapping is expensive; swap minimum value with first element over and over)
- Merge sort (a divide and conquer algorithm; sort two elements each, then sort those sorted groups of two, then four etc.; useful for already previously sorted lists)

For this lab we will be concentrating on quicksort and bubble sort though. The aims of this lab are implementing and comparing sorting algorithms (quicksort and bubble sort) as well as quantifying their algorithmic complexity and to furthermore enhance the quicksort algorithm such that it is effectively more efficient.

The goals of both sorting algorithms are naturally are to sort a list of elements that can range from numbers stored in an array to objects in complex data structures. The steps for the algorithms in very abstract, high-level pseudo code are as follows. (More detail in coming sections)

Bubble sort – High-level explanation:

Iterate through the list 2 elements at a time
If the second is smaller than the first swap them
Repeat everything again until the list is sorted and nothing is swapped anymore

Quicksort – High-level explanation:

Pick a “pivot” from the list
Swap elements such that all values less than the pivot go before it and the greater ones after it
Split the list into two parts at the pivot
Recursively sort these two sub lists until a list only has 1 or less elements

The efficiency of these algorithms will be explored in the pages to come, as well as how a hybrid algorithm can be created that is even more efficient.

Code Design for All Algorithms

For the final complete code listing, see attached C files.

Quicksort – Commented code listing:

```
/* Call by reference to actually change the array and not just a copy. */
/* Faster, uses less memory and is more efficient. */
/* All printf statements are commented out as they skew the timing results. */
void quickSort(int *a, int l, int r) {
        int v,i,j,temp;

        /* If rightmost index is less than or equal to leftmost index, fall back */
        if (r<=l)
                return;

        /* Set right hand value as pivot */
        v=a[r];
        /* Initialise i as leftmost index (-1 as it increments first) */
        i=l-1;
        /* Initialise j as rightmost index */
        j=r;
        //printf("Left sub-array index = %d.\nRight sub-array and pivot index = %d.\nPivot value
a[%d]=%d. \n\n",l,r,r,v);
        //system("PAUSE");
        while(1) {
                /* Look for first value greater than pivot from left to right. */
                while (a[++i]<v);
                /* Look for first value less than pivot from left to right. */
                while (a[--j]>v);
                /* When indicies cross, break out of loop. */
                if (i>=j)
                        break;

                /* Swap leftmost element greater than pivot with rightmost element less than pivot */
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;

                //printarray(a, r+1);
                //system("PAUSE");
        }
        /* Swap first element where indicies cross with pivot to put pivot in the correct place */
        temp=a[i];
        a[i]=a[r];
        a[r]=temp;

        //printarray(a, r+1);
        //system("PAUSE");

        /* Recursion on new sub-arrays */
        quickSort(a,l,i-1);
        quickSort(a,i+1,r);
}
```

Bubble sorting an integer array – Low-level pseudo code:

For all elements in an array except the last
    If an element is greater than the one after it
        Swap the elements using a temporary variable
        Increment a swap counter
Repeat while the number of swaps is greater than 0

Bubble sorting an integer array – C implementation:

```
void bubbleSort(int *a, int first, int size) {
        int i, swapCount, temp;

        do {
                swapCount=0;
                for(i=first;i<(size-1);i++)
                        if(a[i]>a[i+1]) {
                                temp = a[i];
                                a[i] = a[i+1];
                                a[i+1] = temp;
                                swapCount++;
                                //printarray(a, size+1);
                        }
        }while(swapCount>0);
}
```

Enhanced quicksort – Commented code listing:

```
/* This is the number of elements at which quicksort becomes more efficient */
#define algInters 18

void enhancedQuickSort(int *a, int l, int r) {
        int v,i,j,temp;

        /* If sub-array size (rightmost index) is less than 18, bubble sort instead */
        if(r<algInters) {
                bubbleSort(a, l, r);
                return;
        }

        /* If rightmost index is less than or equal to leftmost index, fall back */
        if (r<=l)
                return;

        /* Set median as pivot */
        v=a[(l+r)/2];
        /* Initialise i as leftmost index (-1 as it increments first) */
        i=l-1;
        /* Initialise j as rightmost index */
        j=r;
        //printf("Left sub-array index = %d.\nRight sub-array and pivot index = %d.\nPivot value
a[%d]=%d. \n\n",l,r,r,v);
        //system("PAUSE");
        while(1) {
                /* Look for first value greater than pivot from left to right. */
                while (a[++i]<v);
                /* Look for first value less than pivot from left to right. */
                while (a[--j]>v);
                /* When indicies cross, break out of loop. */
                if (i>=j)
                        break;

                /* Swap leftmost element greater than pivot with rightmost element less than pivot */
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;

                //printarray(a, r+1);
                //system("PAUSE");
        }
        /* Swap first element where indicies cross with pivot to put pivot in the correct place */
        temp=a[i];
        a[i]=a[r];
        a[r]=temp;

        //printarray(a, r+1);
        //system("PAUSE");

        /* Recursion on new sub-arrays */
        enhancedQuickSort(a,l,i-1);
        enhancedQuickSort(a,i+1,r);
}
```

(See results for a full explanation)

Timing the algorithms – Commented code listing:

```
int main(void) {
	int *a, size, i, j;
	double totalTime;
	srand(time(NULL));
	printf("Size(int)\tTime(us)\n");
	printf("------------------------\n\n");
	/* Compute time complexity for many sizes */
	for(size=2;size<=1000;size++) {
		/* Time the same size several times and get an average for accuracy */
		totalTime = 0;
		for(i=0;i<1000;i++) {
			/* Create an array of a specified size */
			a = (int*)malloc(size*sizeof(int));
			/* Initialise it with random numbers */
			for(j=0;j<size;j++)
				a[j] = rand();
			//printf("Unsorted array contents:\n");
			//printarray(a, size);
			timer_start();

			/* Algorithm used */
			//quickSort(a,0,size-1);
			//bubbleSort(a,0,size-1);
			enhancedQuickSort(a,0,size-1);

			timer_stop();
			totalTime += timer_microseconds();
			//printf("Sorted array contents:\n");
			//printarray(a, size);
			free(a);
		}
		printf("%d\t\t%f\n", size, totalTime/1000);
		//system("PAUSE");
	}
	return 0;
}
```

The timing of any algorithm literally works exactly the same way as in section A. Here however, the average estimation is built in; for every size from 1 to 1000, the algorithm is tested 1000 times and the average time calculated. Once the timing is done, the results are printed in table form. Although this can get pretty intensive at times, it's extremely accurate especially since the quicksort code has been optimised to manipulate the array directly.

Results and Conclusions

Quicksort – example output of sorting, for size 10 as an example:

```
Unsorted array contents:
 [28173]  [30560]  [17123]  [6617]  [22700]  [16514]  [3519]  [702]  [3277]  [30
184]

Left sub-array index = 0.
Right sub-array and pivot index = 9.
Pivot value a[9] = 30184.

 [28173]  [3277]  [17123]  [6617]  [22700]  [16514]  [3519]  [702]  [30560]  [30
184]

 [28173]  [3277]  [17123]  [6617]  [22700]  [16514]  [3519]  [702]  [30184]  [30
560]

Left sub-array index = 0.
Right sub-array and pivot index = 7.
Pivot value a[7] = 702.

 [702]  [3277]  [17123]  [6617]  [22700]  [16514]  [3519]  [28173]

Left sub-array index = 1.
Right sub-array and pivot index = 7.
Pivot value a[7] = 28173.

 [702]  [3277]  [17123]  [6617]  [22700]  [16514]  [3519]  [28173]

Left sub-array index = 1.
Right sub-array and pivot index = 6.
Pivot value a[6] = 3519.

 [702]  [3277]  [3519]  [6617]  [22700]  [16514]  [17123]

Left sub-array index = 3.
Right sub-array and pivot index = 6.
Pivot value a[6] = 17123.

 [702]  [3277]  [3519]  [6617]  [16514]  [22700]  [17123]

 [702]  [3277]  [3519]  [6617]  [16514]  [17123]  [22700]

Left sub-array index = 3.
Right sub-array and pivot index = 4.
Pivot value a[4] = 16514.

 [702]  [3277]  [3519]  [6617]  [16514]

Sorted array contents:
 [702]  [3277]  [3519]  [6617]  [16514]  [17123]  [22700]  [28173]  [30184]  [30
560]

Press any key to continue . . .
```

As seen, the algorithm follows the expected steps described in the introduction.

Bubble Sort – example output of sorting, for size 9 (so it's on one line) as an example:

```
Unsorted array contents:
 [31703]  [16947]  [27940]  [1267]  [31635]  [1469]  [25742]  [14814]  [26514]

 [16947]  [31703]  [27940]  [1267]  [31635]  [1469]  [25742]  [14814]  [26514]

 [16947]  [27940]  [31703]  [1267]  [31635]  [1469]  [25742]  [14814]  [26514]

 [16947]  [27940]  [1267]  [31703]  [31635]  [1469]  [25742]  [14814]  [26514]

 [16947]  [27940]  [1267]  [31635]  [31703]  [1469]  [25742]  [14814]  [26514]

 [16947]  [27940]  [1267]  [31635]  [1469]  [31703]  [25742]  [14814]  [26514]

 [16947]  [27940]  [1267]  [31635]  [1469]  [25742]  [31703]  [14814]  [26514]

 [16947]  [27940]  [1267]  [31635]  [1469]  [25742]  [14814]  [31703]  [26514]

 [16947]  [1267]  [27940]  [31635]  [1469]  [25742]  [14814]  [31703]  [26514]

 [16947]  [1267]  [27940]  [1469]  [31635]  [25742]  [14814]  [31703]  [26514]

 [16947]  [1267]  [27940]  [1469]  [25742]  [31635]  [14814]  [31703]  [26514]

 [16947]  [1267]  [27940]  [1469]  [25742]  [14814]  [31635]  [31703]  [26514]

 [1267]  [16947]  [27940]  [1469]  [25742]  [14814]  [31635]  [31703]  [26514]

 [1267]  [16947]  [1469]  [27940]  [25742]  [14814]  [31635]  [31703]  [26514]

 [1267]  [16947]  [1469]  [25742]  [27940]  [14814]  [31635]  [31703]  [26514]

 [1267]  [16947]  [1469]  [25742]  [14814]  [27940]  [31635]  [31703]  [26514]

 [1267]  [1469]  [16947]  [25742]  [14814]  [27940]  [31635]  [31703]  [26514]

 [1267]  [1469]  [16947]  [14814]  [25742]  [27940]  [31635]  [31703]  [26514]

 [1267]  [1469]  [14814]  [16947]  [25742]  [27940]  [31635]  [31703]  [26514]

Sorted array contents:
 [1267]  [1469]  [14814]  [16947]  [25742]  [27940]  [31635]  [31703]  [26514]

Press any key to continue . . .
```

Likewise, the steps described in the introduction for bubble sort are clearly represented here.

Example of quicksort timing output – Similar for any other algorithm (with different numbers):

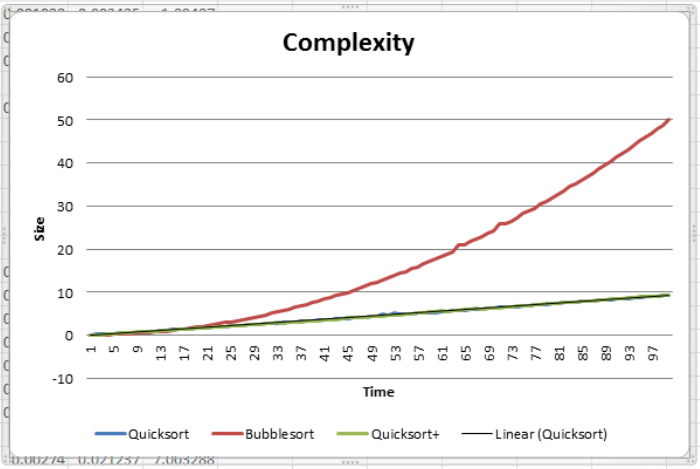
```
Size(int)        Time(us)
-----------------------

2                1.228422
3                1.398229
4                1.378981
5                1.548360
6                1.639893
7                1.876424
8                1.855893
9                2.385843
10               2.072321
11               2.296021
12               2.466682
13               2.921780
14               2.684822
15               2.755824
16               3.041115
17               2.899539
18               2.911087
19               2.976957
20               3.145480
21               3.411524
22               3.470978
23               3.624103
24               4.634814
25               5.784963
26               3.970987
27               4.242163
28               6.562137
29               4.345245
30               4.762275
...
```

The results are then copied into excel and graphed from sizes 2 to 1000, each size a pretty good estimate as it is the average of 1000 attempts.

When all results were thus extracted, the following graphs were obtained (the full excel file has also been submitted):

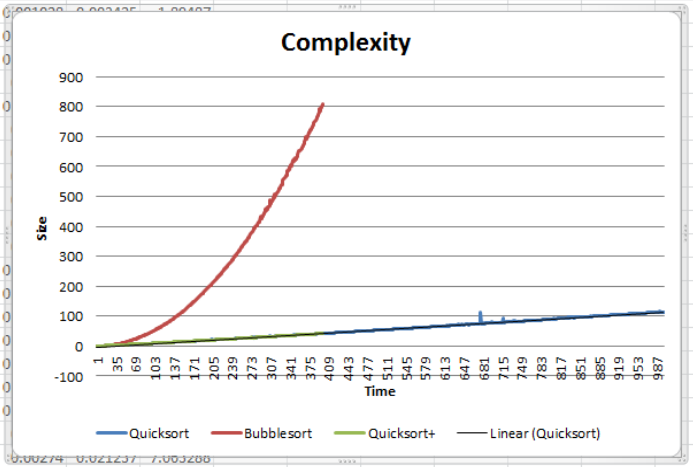
*Graph 1 (small scale) – Complexity for all three algorithms **(axis labels accidentally reversed)**:*

	A	B	C	D	E	F	G	H
1		Size(int)	Quicksort	Bubblesort	Quicksort+	NoAverage		
2		2	0.150028	0.06371	0.105841	0.000343	0.000343	0.002055
3		3	0.216136	0.08529	0.125023	0.000343	0.000685	0.199694
4		4	0.289437	0.109267	0.142835	0	0.001028	0.423366
5		5	0.35897	0.161674	0.323347	0.000343	0.00137	0.659711
6		6	0.450768	0.187706	0.404527	0.000343	0.002055	0.939214
7		7	0.50249	0.242853	0.469607	0.000343	0.002398	1.142676
8		8	0.592575	0.299713	0.553184	0.000685	0.00274	1.38964
9		9	0.657655	0.374384	0.639844	0.000685	0.003083	1.642083
10		10	0.728901	0.458989	0.733354			
11		11	0.841936	0.556609	0.808711			
12		12	0.914552	0.661423	0.873791			
13		13	0.982715	0.765895	0.959766			
14		14	1.059099	0.925856	1.047111			
15		15	1.149184	1.000869	1.117329			
16		16	1.296129	1.154322	1.207757			
17		17	1.308803	1.296129	1.298527			
18	Intersection ->	18	1.41944	1.461913	1.384502			
19		19	1.544805	1.632493	1.460543			
20		20	1.577345	1.873975	1.566042			
21		21	1.659552	1.987695	1.636603			
22		22	1.770532	2.20109	1.756488			
23		23	1.838352	2.420994	1.820541			
24		24	1.960293	2.636445	1.954127			
25		25	2.054831	2.922799	2.004821			
26		26	2.116828	3.108107	2.120254			
27		27	2.187389	3.368771	2.295286			
28		28	2.361051	3.652043	2.283297			
29		29	2.367902	3.918873	2.372012			
30		30	2.455932	4.264827	2.437093	0.002398	0.022607	7.292782
31		31	2.558691	4.47925	2.61418	0.003083	0.022949	7.605853



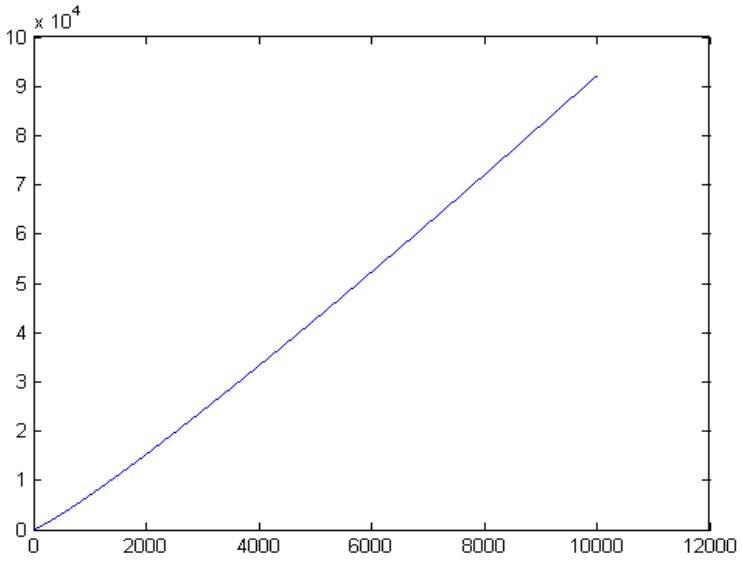
Graph 2 (large scale) – Complexity for all three algorithms:

	A	B	C	D	E	F	G	H
1		Size(int)	Quicksort	Bubblesort	Quicksort+	NoAverage		
2		2	0.150028	0.06371	0.105841	0.000343	0.000343	0.002055
3		3	0.216136	0.08529	0.125023	0.000343	0.000685	0.199694
4		4	0.289437	0.109267	0.142835	0	0.001028	0.423366
5		5	0.35897	0.161674	0.323347	0.000343	0.00137	0.659711
6		6	0.450768	0.187706	0.404527	0.000343	0.002055	0.939214
7		7	0.50249	0.242853	0.469607	0.000343	0.002398	1.142676
8		8	0.592575	0.299713	0.553184	0.000685	0.00274	1.38964
9		9	0.657655	0.374384	0.639844	0.000685	0.003083	1.642083
10		10	0.728901	0.458989	0.733354			
11		11	0.841936	0.556609	0.808711			
12		12	0.914552	0.661423	0.873791			
13		13	0.982715	0.765895	0.959766			
14		14	1.059099	0.925856	1.047111			
15		15	1.149184	1.000869	1.117329			
16		16	1.296129	1.154322	1.207757			
17		17	1.308803	1.296129	1.298527			
18	Intersection ->	18	1.41944	1.461913	1.384502			
19		19	1.544805	1.632493	1.460543			
20		20	1.577345	1.873975	1.566042			
21		21	1.659552	1.987695	1.636603			
22		22	1.770532	2.20109	1.756488			
23		23	1.838352	2.420994	1.820541			
24		24	1.960293	2.636445	1.954127			
25		25	2.054831	2.922799	2.004821			
26		26	2.116828	3.108107	2.120254			
27		27	2.187389	3.368771	2.295286			
28		28	2.361051	3.652043	2.283297			
29		29	2.367902	3.918873	2.372012			
30		30	2.455932	4.264827	2.437093	0.002398	0.022607	7.292782
31		31	2.558691	4.47925	2.61418	0.003083	0.022949	7.605853

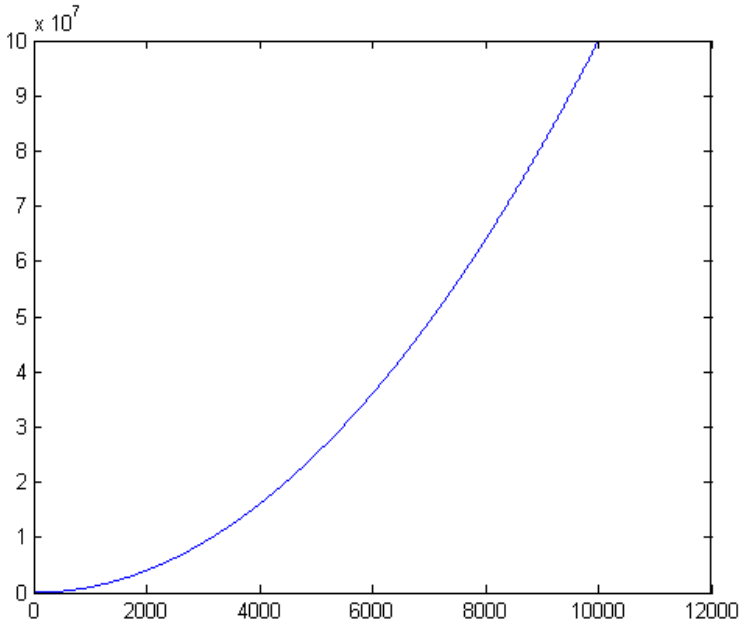


The graphs show that the curve for bubble sort grows in a quadratic manner, while at first glance the standard quicksort curve looks linear. This is however not the case as when plotted against a truly linear line (labelled “Linear (Quicksort)”) one can see the incline in the beginning visible in the small scale graph. The enhanced quick sort takes on the properties of bubble sort below the point of intersection (18) and the properties quicksort beyond that point thus adopting the efficiency of both algorithms suited for different sizes. The curves match the expected results of O(NlogN) and O(N^2) for quicksort and bubble sort respectively:

NlogN



N^2



Clearly for small numbers, bubble sort is quicker than quicksort, so to speak, and vice versa. The improvements did improve the algorithm as evident above and the exact size where the efficiency switches was found to be 18. The enhanced quicksort can therefore be deemed more efficient even if just by a little as one could be working with data structures of any size and the extra step to check if the size is less than 18 (and if so, switch to bubble sort) is negligible by comparison.

Furthermore, selecting median values as a pivot for quicksort is better yet mostly has an equally negligible effect unless scaled up extensively. It simply shifts the curve slightly upwards (see columns to the right of results). Similarly, selecting a pivot value as the median of the last three values barely has any tangible effect. An iterative approach to quicksort is limiting, as the stack size has to be set from the start, and only ever useful if insanely large amounts of data are to be sorted in which case a recursive approach would case a stack overflow. Design-wise, the recursive approach is superior.

End.