MATH49111 Project 1

Rudi Agnew 1013652

November 2021

Abstract

The aim of this project was to code and compare 3 standard sorting algorithms Bubblesort, Quicksort and Heapsort. Using vectors of length n, I graphed the sort times of each algorithms on such vectors as a function of n and analysed the function form of each graph. This was then compared to the theoretical complexity of each algorithm.

1 Creating data to sort

To apply sorting algorithms we need an object to sort, namely vectors. I was given a header file to define class called MVector which represented a vector of double type variables. These vectors were the objects I called the sorting algorithms on to test them. Initially the class contained, along with constructors, 3 member functions. One returned the length of the vector and the other two overloaded the [ ] operator to allow me to access elements of an MVector object using vector notation such as v[1], v[2] etc. However I would add many more member functions as this project progressed. The first 2 of which being functions to swap elements of a vector and overloading the << operator so I could output vectors to the console. I was also given the code to generate a vector with random elements for testing the sorting algorithms on. The full source code of the MVector class can be seen in the appendix A.

2 Bubblesort

2.1 Idea behind Bubblesort

The first and simplest sorting algorithm I looked was called was Bubblesort. Say you have a vector of length n. The idea of Bubblesort is to move the greatest element of that vector to the last position in the vector i.e. position n. You then ignore that last element and look at the vector of length n-1 and move the greatest element of that vector to the position of n-1 and so on. In each vector the method of finding the greatest element is to start at the beginning element and compare it to the element on the right, if the first element is larger it you swap it with the element on the right and then compare it with

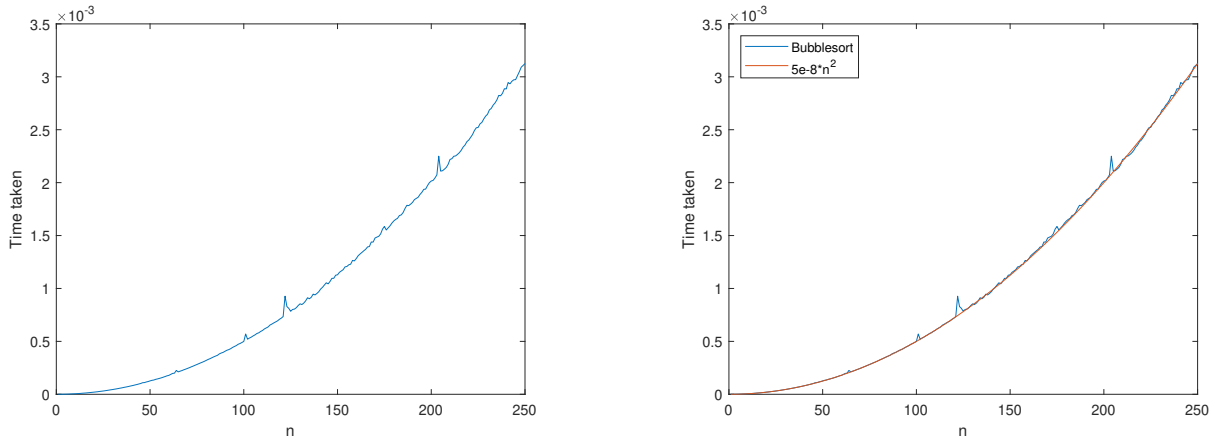
the next element etc. If the first element is smaller it remains in place and we move to the next element and repeat our comparing process. This results in the entire vector being sorted.

2.2 Coding Bubblesort

The parameter of my Bubblesort function was the vector I would be sorting. This was passed by reference as the function changes the vector by sorting it. I then used a **For** loop to go through each element and an **If** statement to check which element was bigger. To reduce the amount of elements I was checking each time I introduced a new variable k and used a **While** loop, incrementing k each time until it was equal to the size of the vector. This **While** loop ran until k was one less than the size of the input vector.

2.3 Sort time analysis for Bubblesort

I ran my Bubblesort function on randomised vectors of length 1 to 250. I did this multiple times per vector to get an average run time. Then, I plotted the time taken for Bubblesort to run as a function of n, the size of the input vector. The resulting graph looks likes some n^2 function and indeed, scaling by 5×10^{-8} we get an almost exact match. This implies that the average time complexity order of Bubblesort is $\mathcal{O}(n^2)$.



(a) Bubblesort time complexity

(b) Comparison

Figure 1: (a) Shows the graph of time taken for Bubblesort against n, (b) is comparing it to an appropriately scaled n^2 function.

Does this make sense? If we look at what is going on in Bubblesort we see that the algorithm does

$(n-1)+(n-2)+(n-3)+...+3+2+1$ comparisons in the loop. Using the fact that

$$\sum_{n=1}^{n-1} k = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$$

we see the order is indeed $\mathcal{O}(n^2)$ and so the graphs make sense. Note this is the same regardless of how sorted the vector was initially as the loops always run through the same number of elements depending on the length of the vector due to how I have coded it. So the best, average and worst case scenarios are all $\mathcal{O}(n^2)$ However, the algorithm can be improved by introducing a monitoring system. If no swaps occur in the **For** loop this means the vector is already sorted and there is no reason to continue. Imagine we ran this optimised Bubblesort on an already sorted vector, the program would just do *n-1* checks i.e $\mathcal{O}(n)$. This is one of the advantages of this optimised Bubblesort that I shall discuss more in section 5.

3 Quicksort

3.1 Idea behind Quicksort

Quicksort is an algorithm that is defined recursively. Initially, we choose a random element from our vector, say x, and then divide the vector into 3 sub-vectors. The first being all elements less than x, second all elements equal to x and third all elements greater than x. We then reapply the algorithm to the first and third sequence and so on.

3.2 Coding Quicksort

To implement Quicksort, I needed to write a couple of new member functions for the MVector class. I needed a way to select a random element of my vector and to then partition accordingly, I first focused on partitioning and left the randomness for later. The partitioning scheme I opted for is called the Lomuto partition scheme and is atributed to Nico Lomuto. The element we are comparing to all other elements to is called a pivot and initially we choose our pivot as the last element in the vector. Note this will not interfere with implementing a random selection later on as we can just swap this randomly chosen element with the last element in the vector before the partitioning. We want all the elements on the left hand side of the pivot to be less than the pivot and all the elements on the right hand side be greater than the pivot. So we introduce two indices, index1 and index2. Index2 is at the location of the pivot and Index1 starts at the first element. We then introduce a **for** loop starting at index1 and running up to index2. We then compare the element at index1 to the last element i.e. the pivot. If the pivot is smaller index1 remains unchanged and we continue looping. If the pivot is bigger we swap the elements at index1 and index2 and then increment index2. This process is repeated for all the elements in the vector up until we

reach the pivot. When we reach the pivot we just then swap it with the element at index1. This process gives us our partitioned vector. I wrote a member function that does this exact process called **partition**.

So now we have a partitioned vector with our pivot somewhere so that all elements to the left are smaller and all elements to the right larger. These are our two sub-vectors we want to reapply our partitioning process to. To do this we want our **partition** function to take in two parameters being the start and end points of these sub-vectors, these parameters are index1 and index2. We want our **partition** function to return the index where the pivot is which is index1, this way we can reapply **partition** to the elements running from this value to the end of the vector and to the elements running from the start of the vector up until this value.

All that's left now is to add randomness and bring everything together. For the randomness I simply used the **rand** function from the standard library. Working modulo (index2-index1) and then adding index1 to that gives will give us the correct range to work in for choosing a pivot. This is because working mod (index2-index1) gives us the correct number of values we can choose from but we need to start at index1 hence why we add it on. We then swap this randomly chosen pivot with index2 and call **partition**. This is all implemented in my **random_partition** function. To finish, I created two new functions **quick_recursive** and **quick**. The first of which takes in 3 arguments, the vector to be sorted/partitioned and the start and end points of this vector. Note if the start and end points of the vector are equal then the algorithm must terminate, this is when the vector is sorted. Hence we use an **If** statement to check that the start point is less than the end point before doing anything. Initially we want to call our **random_partition** function on the whole vector, so the input parameters start and end will be 0 and the size of the vector minus 1 respectively. When we call **random_partition** this partitions the vector and returns the pivot. This is where the recursive element of Quicksort comes in. Now, we want to call **quick_recursive** again but this time once on the sub-vector to the left of the pivot, i.e. with start value 0 and end value equal to the pivot minus 1, and then on the right side of the pivot, i.e. start value pivot plus 1 and end value the length of the vector minus 1. Since our arguments for the start and end are always going to be 0 and the size of the input vector minus 1 we write another 'wrapper' function called **quick** which just takes in the input vector as an argument and calls **quick_recursive**.

3.3 Sort time analysis for Quicksort

I tested my code in the same manner as for Bubblesort however this time the shape of the curve is not immediately clear. Hence I shall deduce the time complexity first and then see if I can match it to the

graph.

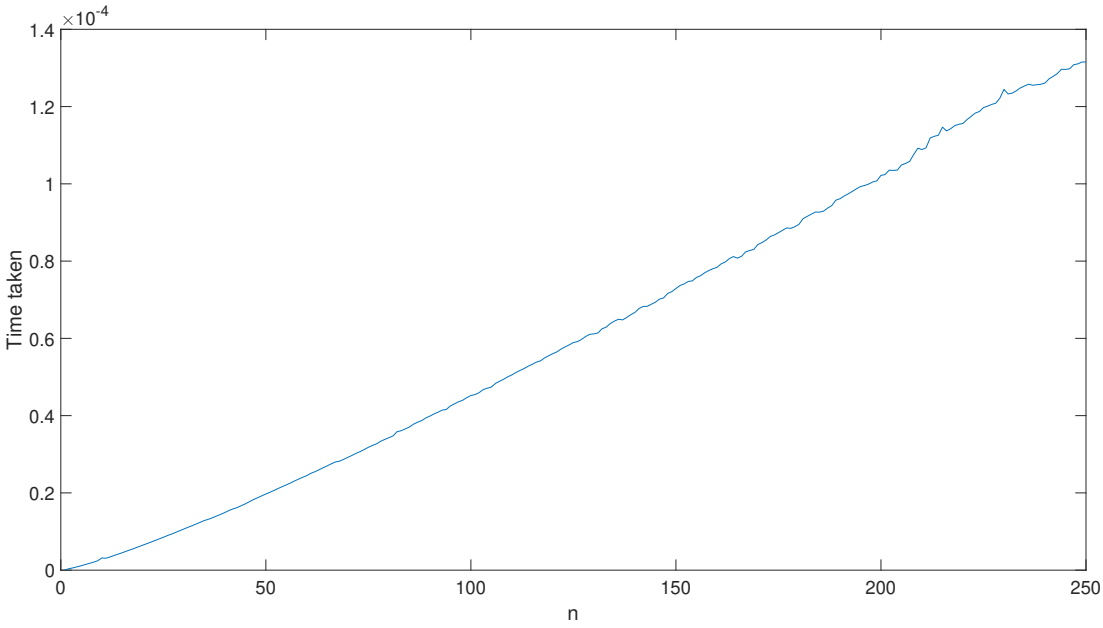


Figure 2: Time taken for Quicksort to run as a function of n.

First note that the order of the partition function is just the size of the input vector, say n. Looking at the case where the pivot ends up in the middle of the vector after the partition we see the vector is then split into two sub-vectors of size $n/2$ each to be then further partitioned. Hence looking at how **quick_recursive** is setup, we have once instance of calling **partition** on a vector of size n and two instances of calling **quick_recursive** on sub-vectors of size $n/2$ hence we see that the time complexity is defined by the following recurrence relation

$$T(n) = \mathcal{O}(n) + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

where T(n) is the time complexity of **quick_recursive**. Solving this recurrence relation we find $T(n) = \mathcal{O}(n\log(n))$. Indeed plotting the time taken divided by n against n the graph is clearly a scaled version of $\log(n)$ so this backs up the theory.

Thinking about a hypothetical scenario where the vector is already sorted and the leftmost element is chosen as the pivot, what happens with the algorithm? First we call partition which is $\mathcal{O}(n)$, we then call partition which since the vector is already sorted, sends the pivot to the first element. Now the recursive function is called again but this time only on the sub-vector of length n-1, it will then be called on a sub-vector length n-2 and so on. So summing this we have $\mathcal{O}(n) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$, the same as Bubblesort. This is the worst case scenario for Quicksort.

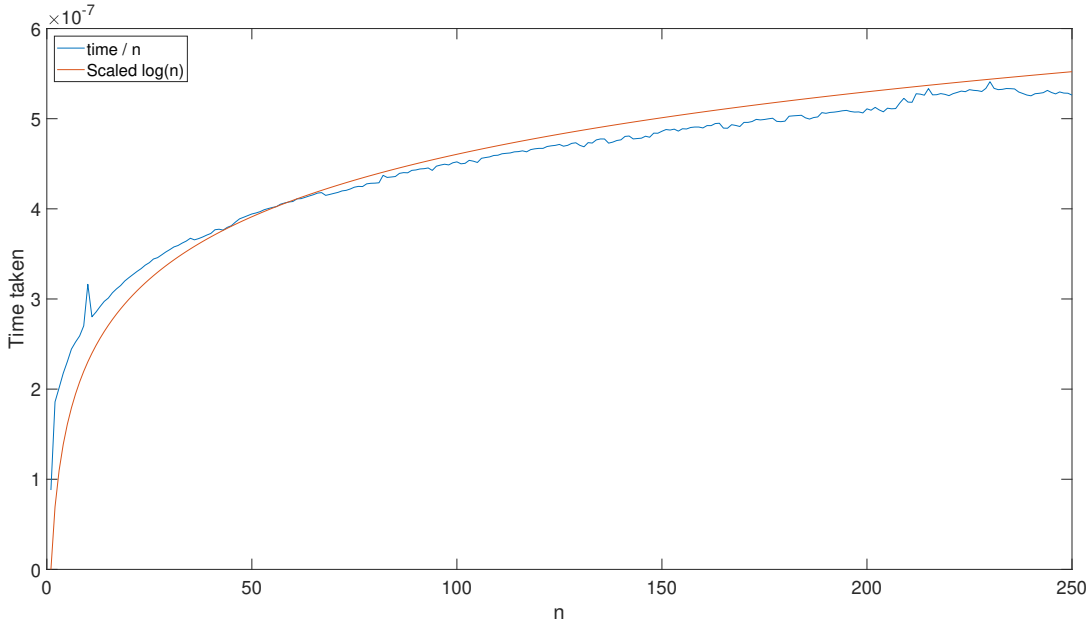


Figure 3: Time taken for Quicksort to run divided by n compared to a scaled logarithmic function.

4 Heapsort

4.1 Idea behind Heapsort

To understand Heapsort one must understand what a binary tree and a heap is. A binary tree is a type of graph which is a set of nodes and edges. The top of node of the tree is called the 'root' and each node has at most two 'child' nodes, see figure 3. A node with no children is called a leaf. A Heap is just a binary tree but with the property that each node is always greater than or equal to its children, see figure 4.

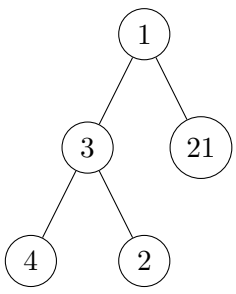


Figure 4: a binary tree

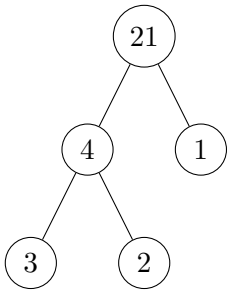


Figure 5: a heap

The Heapsort algorithm takes a binary tree and makes it into a heap. Then, the root of the tree will be the maximum value of the tree. This maximum value is treated as sorted and removed from the tree and replaced with one of the leaf nodes. We then reapply our heap making algorithm to the root node, this

moves the next greatest element to the root and we remove it and so on. Hence this sorts the tree.

4.2 Coding Heapsort

The first task in coding Heasport is to create a binary tree out of a vector and then make it into a heap. To do this we treat the first element of the vector as the root, then the second and third elements as the roots children. The fourth and fifth elements will then be the first and second children of the root's children etc. In other words, for a vector v, the node at v[i] will have its children at v[2i+1] and v[2i+2] while the root is at v[0]. Creating a heap is a bit more tricky and requires recursion.

To do this we look at nodes and check if they are bigger or smaller than their children. If they are bigger we leave them alone, however if they are smaller we swap them with their largest child. If we do this we then need to check that the swap hasn't disrupted the heap structure, so we look at the swapped node and reapply the check, this is the recursive element of the algorithm. If we ignore leaves as they don't have children and start from the bottom of the tree working up, this will create a heap. I created a member function for the MVector class called **is_leaf** returned true if the vertex in question was a leaf and false if not, this will allow us to ignore leaves. The function took two parameters, the first being the size of the vector and the second being the index of the vertex we are checking. To test if the vertex is a leaf or not we just need to check that the value of its children's indices are less than the size of the vector. If they are greater than or equal to it then they cant exist. This is the only additional member function I needed for Heapsort.

Now to actually implement the Heapsort algorithm I wrote two functions, one to build a heap and the other to implement the actual sorting described earlier. To build the heap, **heap_from_root** takes in 3 parameters: The vector acting as the binary tree, the index i being the node acted on and the size of the vector n. First the function calls **is_leaf** on the node at i to check if it is a leaf. If we don't have a leaf we continue and check if the node at i is less than either of its children. If so we then swap the greatest child and recall the function on the swapped child to check if the heap structure is retained. To now implement the sorting algorithm my function **heap** calls **heap_from_root** on all the nodes in the vector using a loop. Once the heap is built, it goes into another loop which first swaps out the root node for a leaf. It then calls **heap_from_root** on the root node but with the size of the vector being 1 less. This then loops until the vector has been sorted.

4.3 Sort time analysis of Heapsort

In a similar fashion to Bubblesort and Quicksort, I graphed my Heapsort algorithm's sort time as a function of vector length n. The shape of the curve was extremely similar to that of Quicksort so once again I divided the sort time by n and indeed the graph has a logarithmic shape. Hence we may conclude that the algorithm is $\mathcal{O}(n\log(n))$.

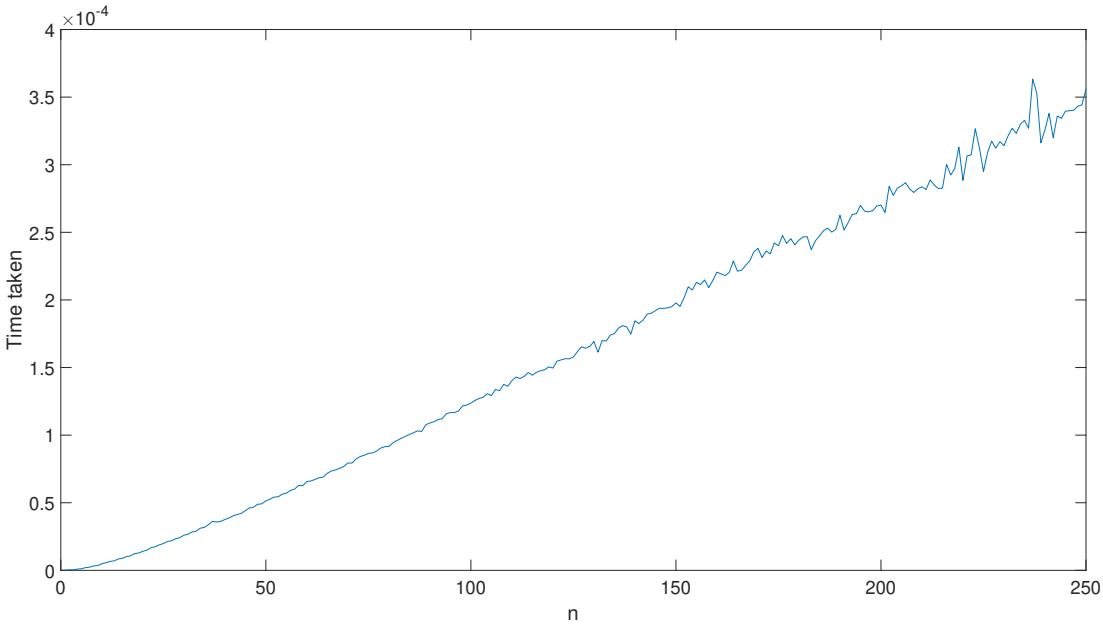


Figure 6: Time taken for Heapsort to run as a function of n.

Does the shape of the curve make sense? If we look at **heap**, this function is calls **heap_from_root** n times. Now, note that every time a new row is introduced the size of the heap increases by a power of 2. For example initially the heap could have a single root, introduce a new row and you have $2 = (2^1)$ more nodes, add another row and you have $4 = (2^2)$ more nodes and so on. Hence the height of this binary tree, or how many rows we have, is no more that $\log_2(n)$, hence we only have to check at most $\log_2(n)$ nodes when calling **heap_from_root** on the root of the heap. Hence this function is $\mathcal{O}(\log(n))$. Bringing it all together we see the algorithm is $\mathcal{O}(n\log(n))$ and thus our graph make sense.

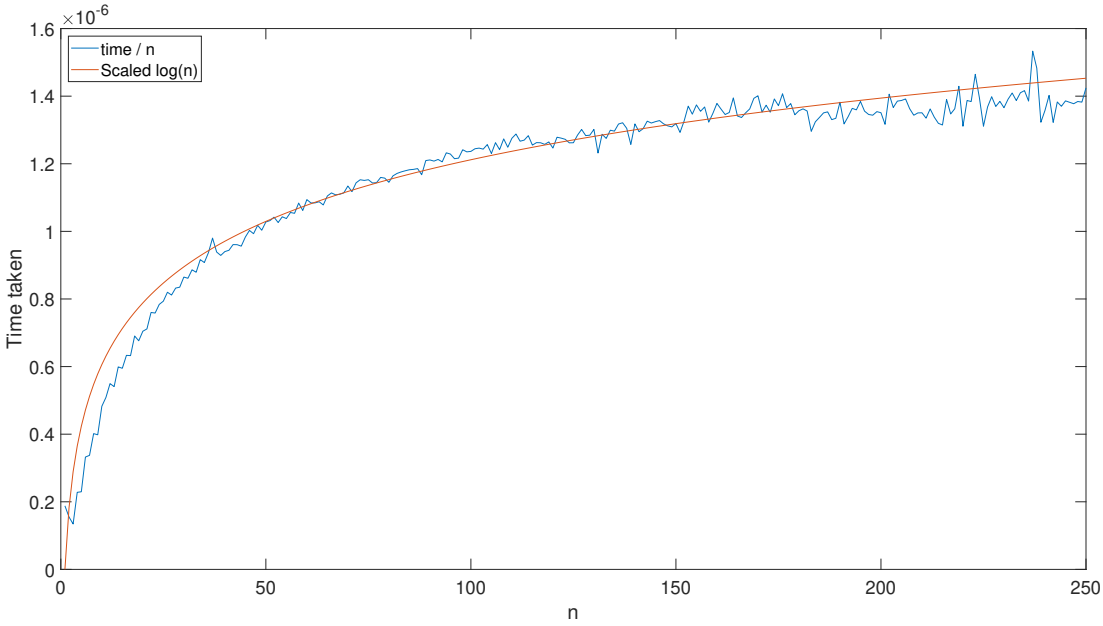


Figure 7: Time taken for Heapsort to run as a function of n compared to a scaled logarithmic function.

5 Comparison and conclusion

One could argue that Heapsort is the best sorting algorithm as it is $\mathcal{O}(n\log(n))$ in each case whereas Quicksort is only $\mathcal{O}(n\log(n))$ on average, $\mathcal{O}(n^2)$ in its worst case and Bubblesort is always $\mathcal{O}(n^2)$. However comparing run times it is clear that Heapsort is slower than Quicksort, why is this? Another thing to consider is the amount of swaps each algorithm does. In the case of Heapsort you are always going to swap at least the number of elements in the vector, this is gauranteed as you have to replace the root with the last leaf each time the loop runs. However for Quicksort this is not the case and often does less sorts that the size of the vector, depending on how the pivot is chosen and how sorted the vector already is. As the vector size gets higher the impact of the number of swaps increases.

A note on Bubblesort, if we introduced the improved algorithm briefly discussed in section 2.3, the best case scenario would be $\mathcal{O}(n)$ which is better than both Heapsort and Bubblesort in their best case scenarios of $\mathcal{O}(n\log(n))$. Also if we compare average sort times for all of the algorithms for small vectors we see they are all almost identical.. One could argue Bubblesort is the superior sorting method for these smaller vectors as it is easier to code, understand and implement compared to the other two, in particular it doesn't require recursion which could be seen as a higher level topic. Despite all this, I think it is sensible to conclude Quicksort as the 'best' sorting algorithm due to its superior speed for larger vectors.

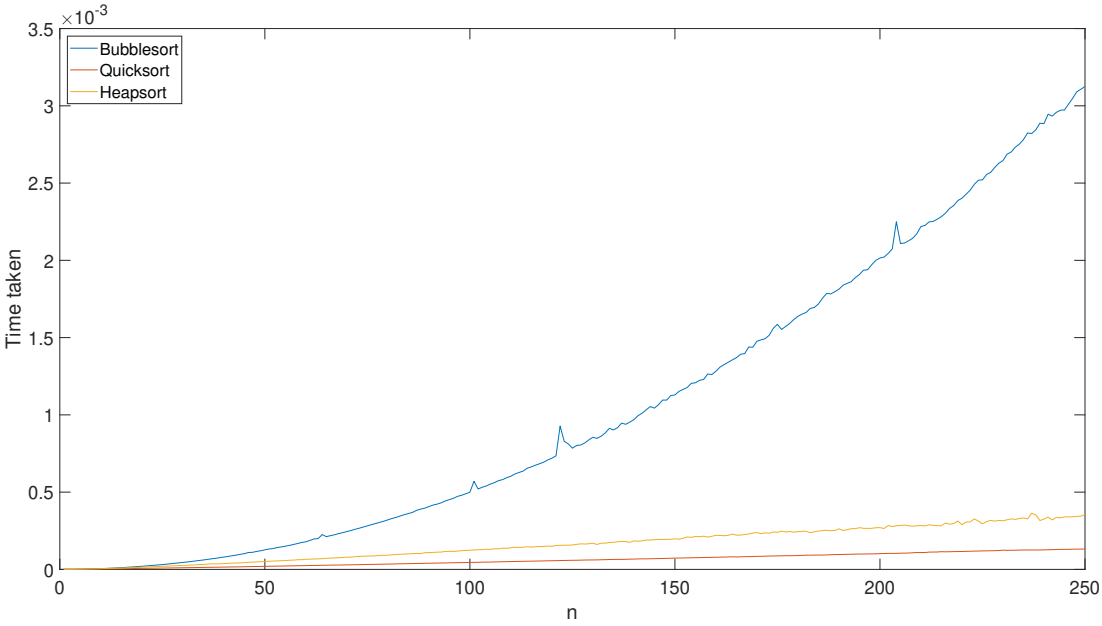


Figure 8: Sort time comparison of the three algorithms as functions of vector size n.

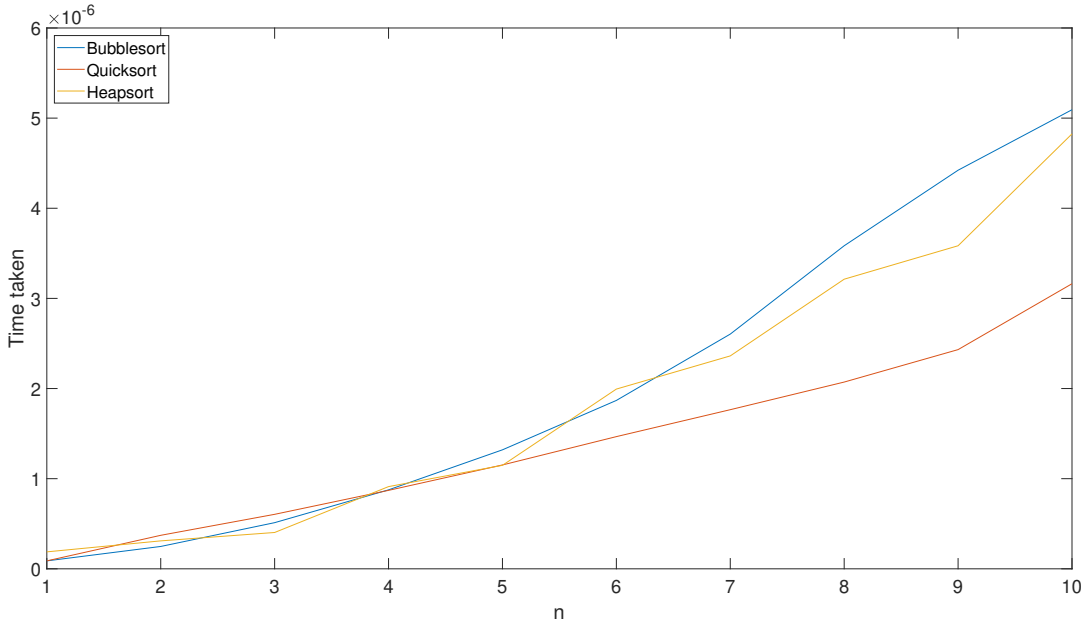


Figure 9: Sort time comparison of the three algorithms as functions of vector size n for small n.

A Source code

Listing 1: MVector class

```
#ifndef MVECTOR_H // the 'include guard'
#define MVECTOR_H
// Class that represents a mathematical vector
class MVector
{
public:
    // constructors
    MVector() {}
    explicit MVector(int n) : v(n) {}
    MVector(int n, double x) : v(n, x) {}
    MVector(std::initializer_list<double> l) : v(l) {}


    // access element and change its value
    double &operator[](int index)
    {
        size_t s = v.size();
        if (index >= s || index < 0)
        {
            std::cout << "Index out of bounds!";
            exit(1); // exit with an error
        }
        return v[index];
    }

    // access element without changing value
    double operator[](int index) const
    {
        size_t s = v.size();
        if (index >= s || index < 0)
        {
            std::cout << "Index out of bounds!";
            return 0; // exit with an error
        }
        return v[index];
    }
```

```
// number of elements
int size() const
{
    return v.size();
}

// swaps elements
void swap(int i, int j)
{
    double temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

// allows use of << on MVector types
friend std::ostream& operator<<(std::ostream &os, const MVector v)
{
    size_t s = v.size();
    os << "["; // brackets for visual aesthetic
    // loops up to and including penultimate entry
    for (size_t i = 0; i < s-1; i++)
    {
        os << v[i] << ", ";
    }
    os << v[s - 1]; // final entry doesnt have a comma after it
    os << "]";
    return os;
}

// assigns an MVector's entries to random numbers between xmin and xmax
void initialise_random(double xmin, double xmax)
{
    size_t s = v.size();
    for (size_t i = 0; i < s; i++)
    {
        v[i] = xmin + (xmax - xmin) * rand() / static_cast<double>(RAND_MAX);
        // need static_cast to get a decimal fraction
```

```
    }

}

// paritions an MVector object into 3 sections [S1,S2,S3] based
// on the last element x. S1 where all elements are less than x,
// S2 where all elements are equal to x and S3 where all elements
// are greater than x.
int partition(int index1,int index2)
{
    // index2 denotes where x (or the 'pivot') is i.e. the element
    // we are comparing everything else in v to.
    // This has to be the last element in the vector.

    // get value at end point
    double x = v[index2];

    // want to loop through elements of v checking if they are
    // greater than, equal to or less than x.
    for (int i = index1; i < index2 ; i++)
    {
        if (v[i] <= x)
        {
            swap(i, index1);
            index1++;
        }
    }
    // when we reach x, swap x with the element at point index2.
    swap(index1, index2);

    // returns where x is located in the vector after the partition
    // so we can then partition to the left and right of it.
    return index1;
}

// chooses a random x, sends it to the last position
// in the vector and then calls partition()
int random_partition(int index1, int index2)
```

```
    {
      int indexr = index1 + rand() % (index2-index1);
      swap(indexr, index2);
      return partition(index1, index2);
    }

    // checks if an element is a leaf
    bool is_leaf(int index, int n)
    {
        // vertex vec[i] has children at vec[i*2+1]
        // and vec[i*2 +2]. So to check if v[i] is a
        // leaf i.e. no children we need to check if
        // its childrens' index's are larger than the
        // size of the vector (n)

        if ((index * 2 + 1) >= n or (index * 2 + 2) >= n)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

private:
    std::vector<double> v;
};

#endif
```

Listing 2: Bubblesort function

```
void bubble(MVector &v)
{
    // initially loop through whole vector
    int k = 1;
    while (k < v.size())
    {
```

```
        for (int i = 0; i < v.size() - k; i++)
        {
            // check if left element is greater than right
            if (v[i + 1] < v[i]) v.swap(i, i + 1);
        }
        // increase k to loop through one less element
        k++;
    }
}
```

Listing 3: quick_recursive

```
//quicksort
void quick_recursive(MVector& v, int start, int end)
{
    if (start < end)
    {
        // get pivot index after first partition
        int index = v.random_partition(start, end);

        // partitions to the left of the pivot
        quick_recursive(v, start, index - 1);

        //partitions to the right of the pivot
        quick_recursive(v, index + 1, end);
    }

}
```

Listing 4: quick

```
// wrapper function for quick_recursive
void quick(MVector &v)
{
    quick_recursive(v, 0, v.size() - 1);
}
```

Listing 5: heap_from_root

```

```

```
void heap_from_root(MVector& v, int i, int n)
{
    // ignore leaves
    if (!v.is_leaf(i,n))
    {
        // check if left child is greater and swap
        if (v[i] < v[2 * i + 1] && v[2 * i + 1] > v[2 * i + 2])
        {
            v.swap(i, 2 * i + 1);
            // call heap_from_root again to check that we still have a heap
            // after swapping
            heap_from_root(v, 2 * i + 1, n);
        }
        // do same for right child
        if (v[i] < v[2 * i + 2] && v[2 * i + 2] > v[2 * i + 1])
        {
            v.swap(i, 2 * i + 2);
            heap_from_root(v, 2 * i + 2, n);
        }
    }
}
```

Listing 6: heap

```
void heap(MVector& v)
{
    // initially build the heap
    for (int i = v.size(); i >= 0; i--)
    {
        heap_from_root(v, i, v.size());
    }
    // swap greatest value out and call
    // heap_from_root on smaller tree
    for (int i = v.size()-1; i >= 0; i--)
    {
        // swap out root for leaf
        v.swap(i, 0);

        // reapply heap_from_root to
```

```
        // the root node
        heap_from_root(v, 0, i);
    }
}
```