# **Scientific Computing Project 1**

**Sorting and Searching**

This report will explore three main search algorithms known as:-

1. Bubblesort.
2. Quicksort.
3. Heapsort.

The aim of this report is to show how to implement these search algorithms along with showing the various optimisations which can be made to them. I will also discuss the sort time of each of the above search functions whilst explaining along the way how it’s the complexity of the algorithm is derived. Provided also are figures and tables to aid your understanding of the sorts mentioned above. The majority of the code I have used can be seen in the appendix of this report.

**Section 1: Bubblesort**

* 1. What is Bubblesort?

Bubblesort is a sorting algorithm which works by sequentially moving through a list that needs to be sorted and compares each pair of adjacent elements and swaps them if they are in the wrong order. Below is an example using the following array of numbers: “7 1 4 9 2”. In order to sort from lowest to highest, the bubblesort algorithm executes as follows:

* 1. A Bubblesort Example

**1st Pass:**

(**7 1** 4 9 2) 🡪 ( **1 7** 4 9 2), The algorithm swaps 7 with 1 and 7 > 1 and so needs to be moved further along.

(1 **7 4** 9 2) 🡪 (1 **4 7** 9 2), We swap 7 and 4 as 7 > 4.

(1 4 **7** **9** 2) 🡪 (1 4 **7 9** 2), We don’t swap 7 and 9 as 7 < 9.

(1 4 7 **9 2**) 🡪 (1 4 7 **2 9**), We swap 2 with 9 as 9 > 2.

**2nd Pass:**

(**1 4** 7 2 9) 🡪 (**1 4** 7 2 9)

(1 **4 7** 2 9) 🡪 (1 **4 7** 2 9)

(1 4 **7 2** 9) 🡪 (1 4 **2 7** 9) We swap here since 7 > 2

(1 4 2 **7 9**) 🡪 (1 4 2 **7 9**)

**3rd Pass:**

(**1 4** 2 7 9) 🡪 (**1 4** 7 2 9)

(1 **4** **2** 7 9) 🡪 (1 **2 4** 7 9) We swap here as 4 > 2

(1 2 **4 7** 9) 🡪 (1 2 **4 7** 9)

(1 2 4 **7 9**) 🡪 (1 2 4 **7** **9**)

We must pass through one more time due to the algorithm requiring a single pass to be completed with 0 swaps.

**4th Pass:**

(**1 2** 4 7 9) 🡪 (**1 2** 4 7 9)

(1 **2 4** 7 9) 🡪 (1 **2 4** 7 9)

(1 2 **4** **7** 9) 🡪 (1 2 **4** **7** 9)

(1 2 4 **7 9**) 🡪 (1 2 4 **7 9**)

* 1. A Bubblesort Implementation

Below is a snippet of code I have used to implement the sort. The code for “MVector.h” can be found in the appendix

//Executes a bubble sort for the specificied MVector

void bubble(MVector &v)

{

//stores the size of the vector

int n = v.size();

//stores the position at which the array has been sorted up until.

int knownSortedPos;

//Executes the loop until the very first index is also sorted

do

{

knownSortedPos = 0;

for (int i = 0; i < n - 1; i++)

{

//Compares two elements of the array

if (v[i + 1] < v[i])

{

//Executes the swap (see MVector.h)

v.swap(i, i + 1);

//Sorted until this point

knownSortedPos = i+1;

}

}

//Once 0 position is sorted, no swaps have been made

n = knownSortedPos;

}

while (n != 0);

}

The above code provides certain optimisations over the regular algorithm in that we can observe that the pass will fill the-largest element and puts it in its final place. (Wikipedia, n.d.)

In addition to this, the code accounts for the scenario where more than one element is placed in their final position in a single pass. During each pass, we observe that every element after the last swap are sorted and no longer need to have comparison made between them. As a result, we can stored the knownSortedPos and reduce the number of comparisons our sort needs to make.

By modifying our main function as follows:

int main()

{

srand(time(NULL));

cout.precision(6);

int n = 5;

ofstream sortResults;

sortResults.open("bubble example.txt");

//if attempt to open file failed, returns 1

if (!sortResults) return 1;

MVector testVector(n);

testVector.initialise\_Random(0, 10);

sortResults << "Bubble(" << n << ")" << endl;

sortResults << "Starting Array: " << testVector << endl;

double startTime = Timer(); // take a time measurement

bubble(testVector);

double endTime = Timer(); // take another time measurement

sortResults << "Sorted Array: " << testVector << endl;

sortResults << "Run Time: " << endTime - startTime << "s" << endl;

return 0;

}

We get the following:

**Bubble**(5)

**Starting Array**: (3.11686, 0.0558489, 6.2859, 5.76647, 9.87976)

**Sorted Array:** (0.0558489, 3.11686, 5.76647, 6.2859, 9.87976)

**Run Time**: 1.91999e-05s

Which, as we can see, displays a sorted version of the starting array.

* 1. Observing Bubblesort Algorithm’s run-time

Below is a table showing how bubblesort’s run-time (in seconds) changes as the number of elements it must sort (, increases.

Table 1: Run-Time of Bubble Sort

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | 0.000120831 | 0.00455242 | 0.0019781 | 0.443231 | 1.76913 |
|  | 0.000216831 | 0.00347364 | 0.0019438 | 0.466711 | 1.7012 |
|  | 0.000214783 | 0.00325912 | 0.00203237 | 0.389511 | 1.76606 |
|  | 0.000200447 | 0.00480842 | 0.00201343 | 0.401999 | 1.63567 |
|  | 0.000230655 | 0.00526384 | 0.00111052 | 0.424739 | 1.66933 |
|  | 0.000227583 | 0.00590307 | 0.00139545 | 0.361428 | 1.52955 |
|  | 0.0000788475 | 0.00373655 | 0.00148249 | 0.506405 | 1.68768 |
|  | 0.000208383 | 0.0040842 | 0.00113459 | 0.385979 | 1.64885 |
|  | 0.000142847 | 0.00372964 | 0.00111974 | 0.380105 | 1.62378 |
|  | 0.000174079 | 0.00484016 | 0.00113331 | 0.419234 | 1.71597 |
|  | 0.000181529 | 0.00455242 | 0.00153438 | 0.417934 | 1.67472 |

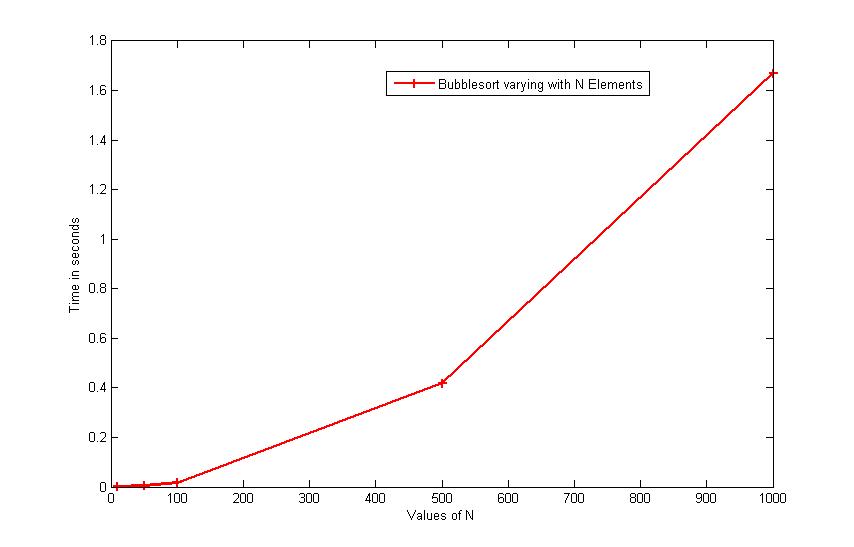


Figure 1: Graph of sort time vs. increasing number of elements

The graph looks similar to the function where is the number of elements in the array.

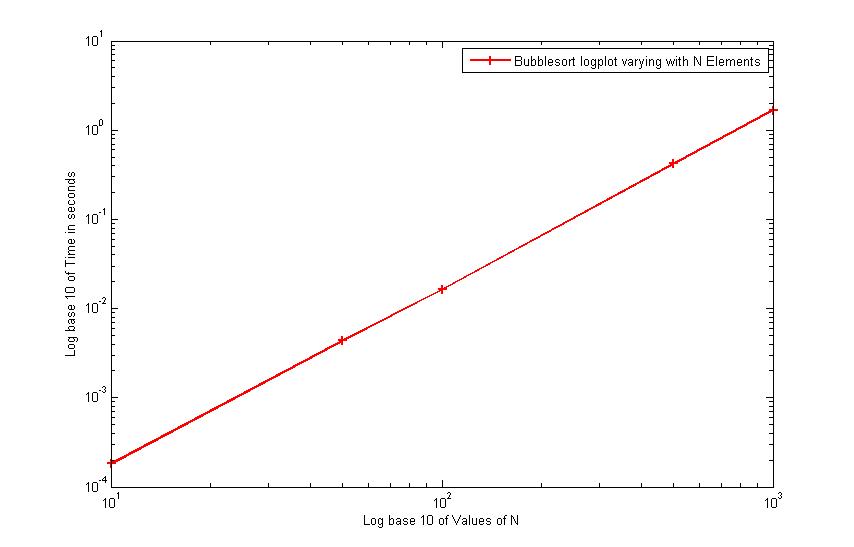


Figure 2: Log plot of sort time vs. increasing number of elements

By calculating the gradient of the log plot, we see that the bubblesort has time-complexity of :

In terms of powers, we see:

Showing that the graph must be of .

* 1. Analysing Bubblesort Algorithm’s Complexity

The space complexity measures how much memory an algorithm requires. As such, the space complexity for bubble sort is simply the first allocation of elements to the array and therefore the space complexity must be .

Bubble sort has worst and average complexity of . The inner loop is executed times and so has complexity . The worst case involves the list being in reverse order i.e. from largest to smallest. Then the smallest element must be swapped in every iteration and so the complexity is of .

The best case is when the list is already sorted. The complexity is then just . No swaps will be performed and the code will only produce 1 pass.

**Section 2: QuickSort**

* 1. What is Quicksort?

Quicksort is another sorting algorithm. The algorithm uses a “pivot” which is an element from the array of values. If the other array values are less than pivot’s value it would be sorted so that the elements in the array whose values are smaller than the pivot point’s value would be placed below the pivot point. The same holds true for elements whose value is larger than the pivot’s value, only that they will be placed above the pivot point. This process repeats until the pivot achieves its final position. This entire process partitions the array into two and is hence called the partition operation.

The steps are: (Quicksort, n.d.)

1. Pick an element, called a **pivot**, from the array.
2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. [Recursively](https://en.wikipedia.org/wiki/Recursion_(computer_science)) apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
   1. A Quicksort Example

We have an array of values, (5 3 7 6 2 1 4) and we are trying to sort using quicksort. The following is the execution of the algorithm:

(5 3 7 6 2 1 4) 5> 4 so swap with the number 1

(1 3 7 6 2 5 4) swapped so advance from both sides

(1 3 7 6 2 5 4) 3 < 4 so don’t swap

(1 3 7 6 2 5 4) 7 > 4 swap with 2

(1 3 2 6 7 5 4) swapped both so advance from both sides

(1 3 2 6 7 5 4) As the indexs cross, the pivot swaps with the index which hols the value 6

(1 3 2 4 7 5 6)

Now apply the same sort to the subarrays: (1 3 2) and (7 5 6).

2.3 A Quicksort Implementation

Below is a snippet of code I have used to implement the quicksort.

void quicksort(MVector &v, int start, int end)

{

int left = start, right = end;

double pivot = v[(start + end)/2];

//Prevent sorting of empty vector

if (end == start)

return;

//partition the vectors

while (left <= right) {

while (v[left] < pivot)

left++;

while (v[right] > pivot)

right--;

if (left <= right) {

v.swap(left, right);

left++;

right--;

}

}

//recursively sort

if (start < right)

quicksort(v, start, right);

if (left < end)

quicksort(v, left, end);

}

With the following wrapped method:

//Wrapper method which calls quicksort

void quick(MVector &v) {

quicksort(v, 0, v.size() - 1);

}

The above code also provides certain optimisations over other algorithms. This is an in-place quicksort which swaps array values to avoid allocating more memory for additional arrays. In the case of an empty array being fed into quicksort, the function returns as there are no elements to sort. It recursively sorts the array through the use of the two sub arrays

By modifying our main function as follows:

int main()

{

srand(time(NULL));

cout.precision(6);

int n = 5;

ofstream sortResults;

sortResults.open("quick example.txt");

//if attempt to open file failed, returns 1

if (!sortResults) return 1;

MVector testVector(n);

testVector.initialise\_Random(0, 10);

sortResults << "Quick(" << n << ")" << endl;

sortResults << "Starting Array: " << testVector << endl;

double startTime = Timer(); // take a time measurement

quick(testVector);

double endTime = Timer(); // take another time measurement

sortResults << "Sorted Array: " << testVector << endl;

sortResults << "Run Time: " << endTime - startTime << "s" << endl;

return 0;

}

We get the following:

**Quick**(5)

**Starting Array**: (0.191351, 5.86444, 7.9928, 7.56096, 5.81591)

**Sorted Array**: (0.191351, 5.81591, 5.86444, 7.56096, 7.9928)

**Run Time:** 2.68798e-05s

Which produced the sorted version of the starting array.   
  
2.4. Observing Quicksort Algorithm’s Runtime

Below is a table showing how quicksort’s’ run-time (in seconds) changes as the number of elements it must sort (, increases.

Table 2: Run-Time of Quicksort

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | 8.67835e-05 | 0.000494333 | 0.0019781 | 0.0093186 | 0.0182849 |
|  | 0.000119295 | 0.000510205 | 0.0019438 | 0.0115233 | 0.0167397 |
|  | 6.29756e-05 | 0.000532733 | 0.00203237 | 0.00783407 | 0.0165498 |
|  | 6.04156e-05 | 0.000534013 | 0.00201343 | 0.00773141 | 0.0195237 |
|  | 6.60476e-05 | 0.000477181 | 0.00111052 | 0.00757832 | 0.0200534 |
|  | 6.52796e-05 | 0.000481021 | 0.00139545 | 0.00819502 | 0.0183989 |
|  | 7.03996e-05 | 0.000540157 | 0.00148249 | 0.00773832 | 0.0180978 |
|  | 6.91196e-05 | 0.000503037 | 0.00113459 | 0.0121085 | 0.0173659 |
|  | 6.60476e-05 | 0.000676604 | 0.00111974 | 0.0118786 | 0.019844 |
|  | 9.93274e-05 | 0.000481533 | 0.00113331 | 0.0101811 | 0.0216145 |
|  | 7.65691e-05 | 0.000523082 | 0.00153438 | 0.00940871 | 0.0186473 |

The graph looks similar to the function where is the number of elements in the array. This can be seen by noting the small curvature near the origin which isn’t present in

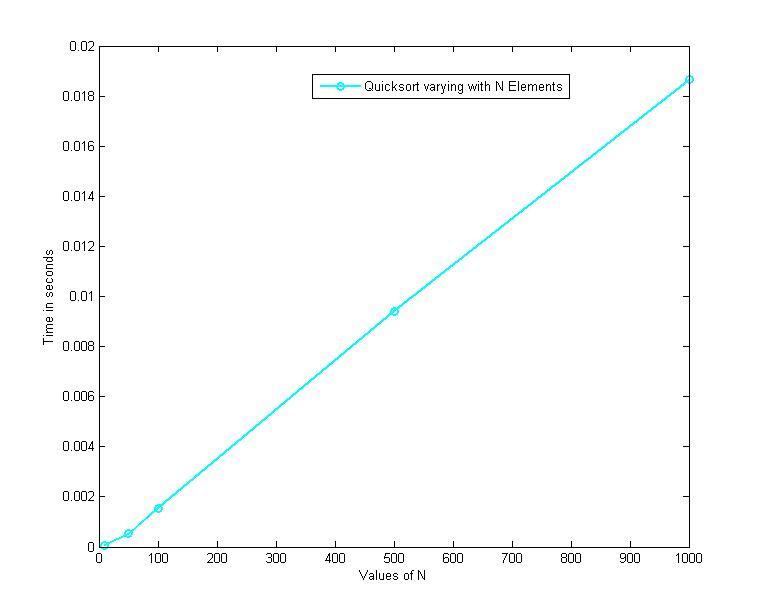


Figure 3: Graph of quicksort time vs. increasing number of elements

By calculating the gradient of the log plot below, we see that the quicksort’s time-complexity function contains :

In terms of powers, we see that the gradient is equal to:

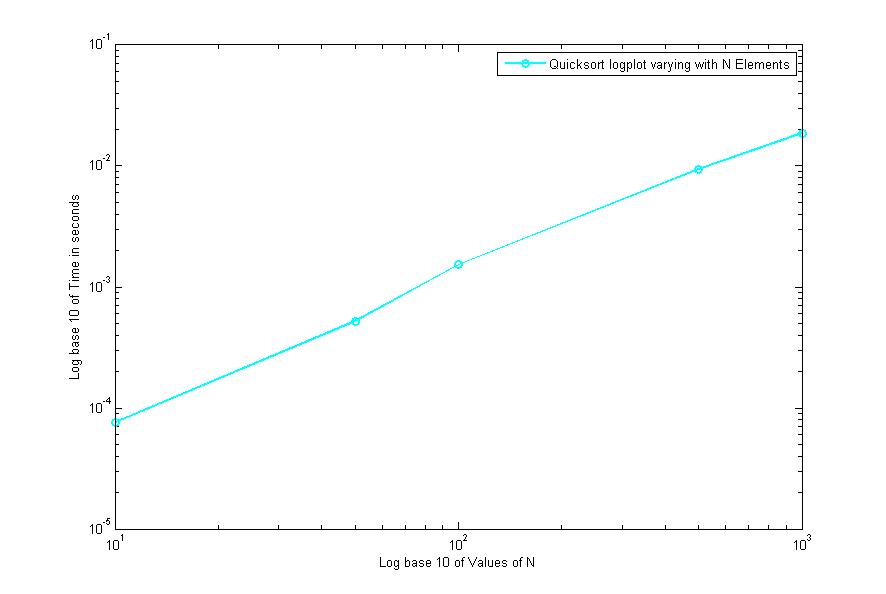
Which gives us the power of the n component in the graph.

Figure 4: logplot of quicksort time vs. increasing number of elements

* 1. Analysing Quicksort Algorithm’s Complexity

The worst case scenario is as follows:

If the pivot chosen by the partition function is always either the largest or the smallest element in the -element subarray, then one of the partitions will contain no elements and the other partition will contain elements i.e. all the elements but the pivot. As a result, the recursive calls on the sub arrays will be of size 0 and respectively.

The first call of quicksort takes time for some constant For the largest recursive calls, it would take as follows: (Algorithms - Quicksort, n.d.)

).

The best case scenario is when both partitions contain close to equal amount of elements i.e. within 1 of each other or equal. If two partitions contain the same number of elements then the sub arrays must both have odd elements and the pivot must be central after the partitioning with each partition having elements. When the sub arrays number of elements are within one of each other, then the original array must have been even – say of length n. One of the partitioning arrays must be of length and the other, As both are less than or equal to we can calculate the best case scenario as follows:

For the first partition: ,

For the 2nd : .

For the 3rd :

As a result, the time-complexity is ).

**Section 3: Heapsort**

* 1. What is Heapsort?

A heap is a type of binary tree where each node contains a value which is always larger than any of its child nodes. A heap sort is divided into 2 parts: building a heap and sorting it.

To build a heap out of data, we place the data in an array in the layout of a binary tree. Each array index refers to a node number. In the second step, we sort the array by repeatedly removing the largest element from the top of the heap and inserting it into the sorted array. Once every node has been removed from the heap, you have performed a heapsort.

* 1. A Heapsort Example

Let the array: **(4 3 1 6 2 5 0)**  be our binary tree.

Running the algorithm presents us with the following:

**(4 3 1 6 2 5 0)** 🡪 **(4 3 5 6 2 1 0)**

**(4 3 5 6 2 1 0) 🡪 (4 6 5 3 2 1 0)**

**(4 6 5 3 2 1 0) 🡪 (6 4 5 3 2 1 0)**

As 6 is the largest element, we swap it with the last indexed element (0) and sort the array of size which in this case is 6 leaving us with: **(0 4 5 3 2 1).**

Repeating this process recursively results in the sorted array of **(6 5 4 3 2 1).**

* 1. A Heapsort Implementation

Below is the code for my own implementation of heapsort.

//Generate the heap

void heap\_from\_root(MVector &v, int i, int n){

int child1, child2, largest;

child1 = (2 \* i) + 1;

child2 = child1 + 1;

//check if inbounds and if its bigger than parents

if (child1 <= n && (v[child1] > v[i]))

largest = child1;

else//if not, set parent as biggest

largest = i;

//compare child2 with current largest

if (child2 <= n && (v[child2] > v[largest]))

largest = child2;

//if the largest isn’t the parent aka it’s a child

if (largest != i)

v.swap(i, largest);

//we’re at the root node

if (i == 0) {}

if (i > 0)

heap\_from\_root(v, i - 1, n);

}

//Sorts the heap

void heap(MVector &v)

{

//start at the lastchild’s parent node

int startIndex = floor((v.size() - 2) / 2);

//generate the heap

heap\_from\_root(v, startIndex, v.size() - 1);

int j;

//Remove largest and repeat.

for (j = v.size() - 1; j >= 1; j--)

{

//swap root with final node

v.swap(0, j);

startIndex = floor((j - 2) / 2);

heap\_from\_root(v, startIndex, j - 1);

}

}

We start by recursively defining the heap and comparing the child to the parents nodes. The parent node is then compared with the two (or 1) child nodes. If the child node holds a larger value, they swap. This is repeated until the largest value is now at the root. The for loop in void heap(MVector &v) removes the largest element from the array by swapping it with the end node and, by reducing the value it inputs as the length of the vector, the heap then sorts as if the last vector position didn’t exist.

By modifying our main function as follows:

int main()

{

srand(time(NULL));

cout.precision(6);

int n = 5;

ofstream sortResults;

sortResults.open("heap example.txt");

//if attempt to open file failed, returns 1

if (!sortResults) return 1;

MVector testVector(n);

testVector.initialise\_Random(0, 10);

sortResults << "Heap(" << n << ")" << endl;

sortResults << "Starting Array: " << testVector << endl;

double startTime = Timer(); // take a time measurement

heap(testVector);

double endTime = Timer(); // take another time measurement

sortResults << "Sorted Array: " << testVector << endl;

sortResults << "Run Time: " << endTime - startTime << "s" << endl;

return 0;

}

We obtain:

**Heap**(5)

**Starting Array**: (4.90341, 4.42915, 3.68938, 7.82067, 8.41731)

**Sorted Array**: (3.68938, 4.42915, 4.90341, 7.82067, 8.41731)

**Run Time**: 5.27357e-05s

As we can see, the above array is clearly sorted.

3.4 Observing Heapsort Algorithm’s Runtime

Below is a table showing how heapsort’s run-time (in seconds) changes as the number of elements it must sort (, increases.

Table 3: Run-Time of Heapsort

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | 0.000245759 | 0.0028828 | 0.00648342 | 0.07732 | 0.31353 |
|  | 0.000131839 | 0.00248626 | 0.00517859 | 0.0794924 | 0.323191 |
|  | 0.000182271 | 0.00170418 | 0.0059727 | 0.0848453 | 0.289653 |
|  | 0.000255486 | 0.00163301 | 0.0057871 | 0.149618 | 0.28398 |
|  | 0.000201471 | 0.00171826 | 0.00647855 | 0.113225 | 0.348716 |
|  | 0.000231935 | 0.00164863 | 0.00618467 | 0.0811259 | 0.277822 |
|  | 0.000269566 | 0.00166681 | 0.00662115 | 0.085526 | 0.27457 |
|  | 0.000225023 | 0.00166553 | 0.0065318 | 0.0827717 | 0.331621 |
|  | 0.000222719 | 0.00164402 | 0.00631062 | 0.0835937 | 0.279004 |
|  | 0.000224767 | 0.00164735 | 0.00682262 | 0.0822472 | 0.275899 |
|  | 0.000219084 | 0.00186968 | 0.00623712 | 0.0919765 | 0.299799 |

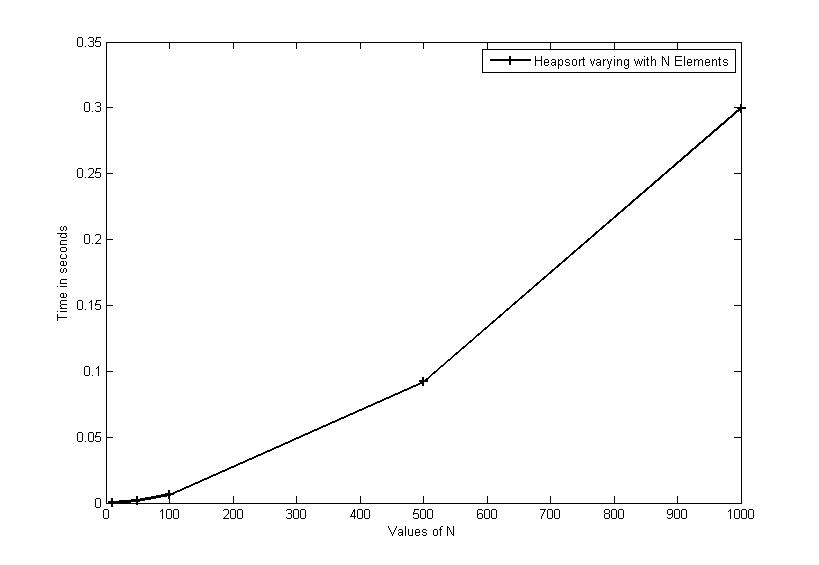
****

Figure 5: Graph of heapsort time vs. increasing number of elements

The graph looks similar to the function where is the number of elements in the array (similar to quick sort).This can be seen by noting the curvature near the origin which isn’t present in

By calculating the gradient of the log plot below, we see that the heapsort’s time-complexity function contains (similar to that of quicksort)

In terms of powers, we see that the gradient is equal to:

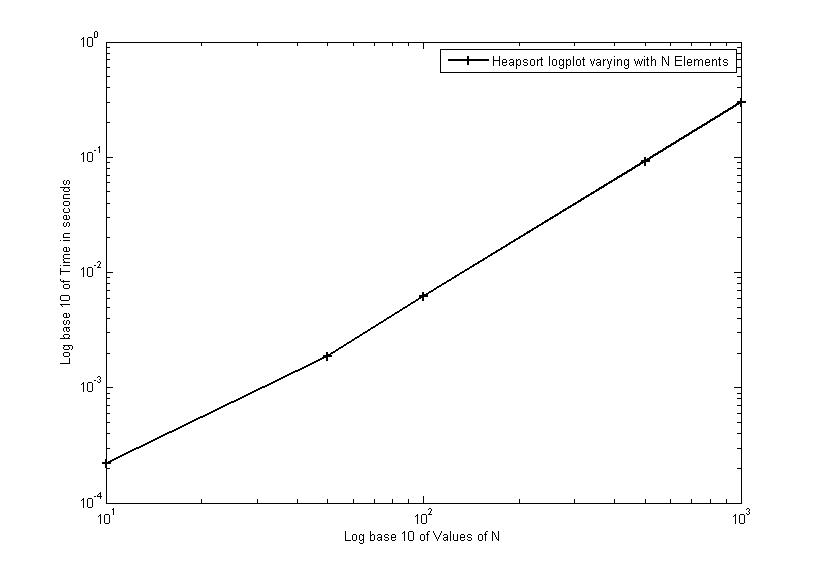
****

Figure 6: logplot of heapsort time vs. increasing number of elements

* 1. Analysing Heapsort Algorithm’s Complexity

Similar to quicksort, heapsort has a complexity of .

On further analysis (Complexity of Heapsort, n.d.), we find the following:

1. The cost of building a heap is .
2. The cost of “Heapify” is .

“The basic heap operation of “Heapify” runs in time, because the heap has levels, and the element being sifted moves down/up one level of the tree after a constant amount of work.

Based on this we can see that:

(1) that it takes time to build a heap, because we need to apply Heapify roughly times (to each of the internal nodes), and

(2) that it takes time to extract each of the maximum elements, since we need to extract roughly n elements and each extraction involves a constant amount of work and one Heapify. Therefore the total running time of HeapSort is ).” (Heapsort Analysis and Partitioning, 1998)

1. Comparing Bubblesort, Quicksort and Heapsort

By observing my results and the theoretical best and worst-case scenario, the end-result is that all the above sorts have their uses.

Whilst bubblesort is slow for large arrays, it can be faster for smaller datasets and also doesn’t take many operations for it to sort a nearly-sorted array whereas quicksort would continue its recursive definition.

If given a random dataset to sort, the time-complexities of each algorithms suggest that quicksort and heapsort are equally as good in the average case with a time complexity of . However, quicksort usually executes the faster of the two and has better average performance. At the same time, quicksort has the downside of having a worst-case of and as such, should not be used in place of heapsort in software that requires a constant (with no sharp variance) in response time i.e. spacecraft. If n is large, you would be better off using heapsort for its guaranteed time.

Bubble and quicksort are both easier to program than heapsort and quicksort also has an elegant recursive definition.

For simplicity and ease of coding, I would choose bubblesort.

For consistency (in guaranteeing time) I would choose heapsort.

However, for most other tasks, due to the better average performance of quicksort, I would rank it above heapsort and heapsort about bubblesort due to its guarantee of time complexity.

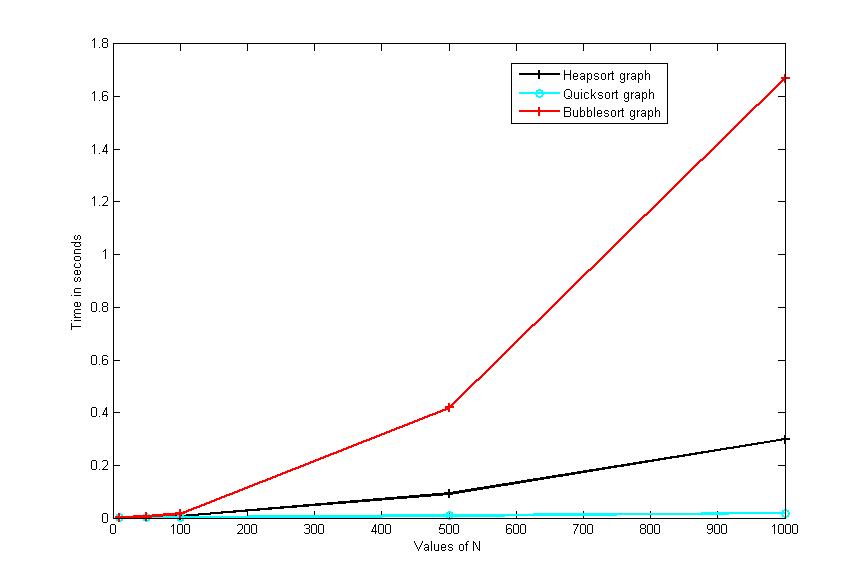


Figure 7: graph of all the sorts: runtime vs. increasing number of elements

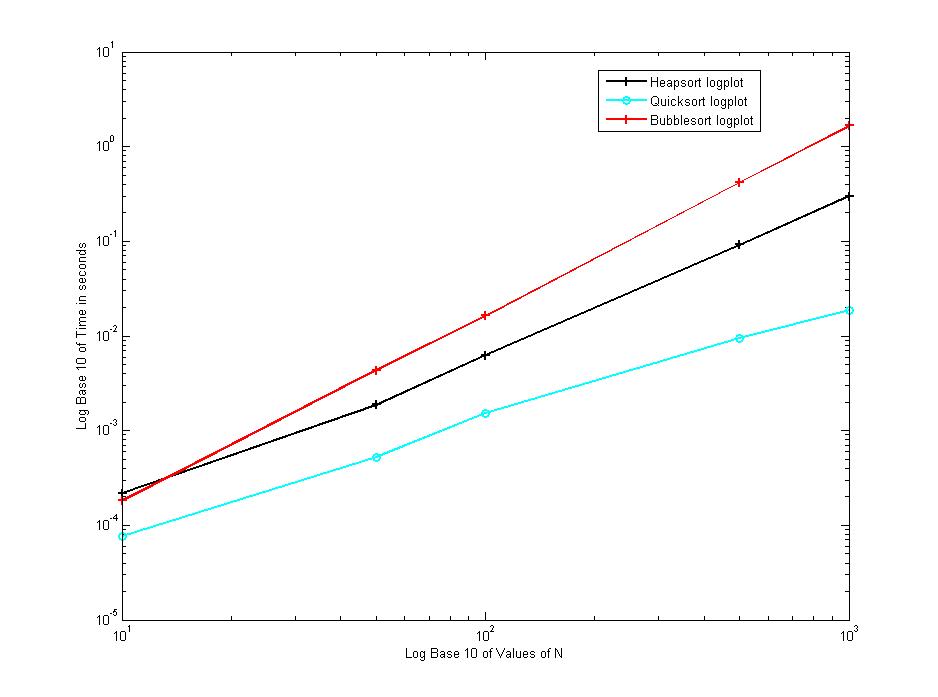
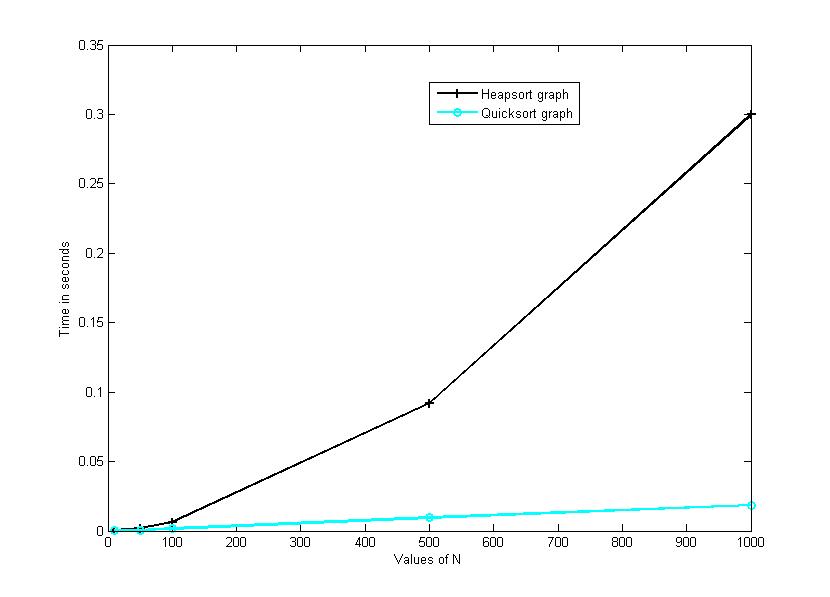


Figure 7: graph of Quick & Heap: runtime vs. increasing number of elements

Figure 8: logplot of all the sorts: runtime vs. increasing number of elements

**Appendix**

**Function bubble:**

//Executes a bubble sort for the specificied MVector

void bubble(MVector &v)

{

//stores the size of the vector

int n = v.size();

//stores the position at which the array has been sorted up until.

int knownSortedPos;

//Executes the loop until the very first index is also sorted

do

{

knownSortedPos = 0;

for (int i = 0; i < n - 1; i++)

{

//Compares two elements of the array

if (v[i + 1] < v[i])

{

//Executes the swap (see MVector.h)

v.swap(i, i + 1);

//Sorted until this point

knownSortedPos = i+1;

}

}

//Once 0 position is sorted, no swaps have been made

n = knownSortedPos;

}

while (n != 0);

}

**Function quickrecursive:**

void quicksort(MVector &v, int start, int end)

{

int left = start, right = end;

double pivot = v[(start + end)/2];

//Prevent sorting of empty vector

if (end == start)

return;

//partition the vectors

while (left <= right) {

while (v[left] < pivot)

left++;

while (v[right] > pivot)

right--;

if (left <= right) {

v.swap(left, right);

left++;

right--;

}

}

//recursively sort

if (start < right)

quicksort(v, start, right);

if (left < end)

quicksort(v, left, end);

}

**Function heap\_from\_root:**

//Generate the heap

void heap\_from\_root(MVector &v, int i, int n) {

int child1, child2, largest;

child1 = (2 \* i) + 1;

child2 = child1 + 1;

//check if inbounds and if its bigger than parents

if (child1 <= n && (v[child1] > v[i]))

largest = child1;

else//if not, set parent as biggest

largest = i;

//compare child2 with current largest

if (child2 <= n && (v[child2] > v[largest]))

largest = child2;

//if the largest isn’t the parent aka it’s a child

if (largest != i)

v.swap(i, largest);

//we’re at the root node

if (i == 0) {}

if (i > 0)

heap\_from\_root(v, i - 1, n);

}

**Function heap:**

//Sorts the heap

void heap(MVector &v)

{

//start at the lastchild’s parent node

int startIndex = floor((v.size() - 2) / 2);

//generate the heap

heap\_from\_root(v, startIndex, v.size() - 1);

int j;

//Remove largest and repeat.

for (j = v.size() - 1; j >= 1; j--)

{

//swap root with final node

v.swap(0, j);

startIndex = floor((j - 2) / 2);

heap\_from\_root(v, startIndex, j - 1);

}

}

**SortableContainer.h:**

using namespace std;

#include <vector>

class SortableContainer

{

public:

// access element (lvalue)

virtual double &operator[](int index) = 0;

// access element (rvalue)

virtual double operator[](int index) const = 0;

virtual int size() const = 0;

virtual void swap(int a, int b) = 0;

virtual void initialise\_Random(double xmin, double xmax) = 0;

virtual bool cmp(int i, int j) = 0;

};

**CoordinateArray.h:**

#include <iostream>

#include "SortableContainer.h"

#include <vector>

using namespace std;

struct IntegerCoordinate {

unsigned X, Y;

};

class CoordinateArray : public SortableContainer {

public:

unsigned getX(int index);

unsigned getY(int index);

void setX(int index, unsigned b);

void setY(int index, unsigned b);

bool CoordinateArray::cmp(int i, int j);

private:

std::vector<IntegerCoordinate> v;

};

unsigned CoordinateArray::getX(int index) {

return v[index].X;

}

unsigned CoordinateArray::getY(int index) {

return v[index].Y;

}

void CoordinateArray::setX(int index, unsigned b) {

v[index].X = b;

}

void CoordinateArray::setY(int index, unsigned b) {

v[index].Y = b;

}

bool CoordinateArray::cmp(int i, int j) {

if (getX(i) < getX(j)) return true;

else if ((getX(i) == getX(j)) && (getY(i) < getY(j))) return true;

else return false;

}

**Function bubble with SortableContainer:**

//Executes a bubble sort for the specificied MVector

void bubble(SortableContainer \*)

{

//stores the size of the vector

int n = v->size();

//stores the position at which the array has been sorted up until.

int knownSortedPos;

//Executes the loop until the very first index is also sorted

do

{

knownSortedPos = 0;

for (int i = 0; i < n - 1; i++)

{

//Compares two elements of the array

if (v->cmp(i + 1, i))

{

//Executes the swap (see MVector.h)

v->swap(i, i + 1);

//Sorted until this point

knownSortedPos = i+1;

}

}

//Once 0 position is sorted, no swaps have been made

n = knownSortedPos;

}

while (n != 0);

}

**Function quickSort with SortableContainer:**

void quicksort(SortableContainer \*v, int start, int end)

{

int left = start, right = end;

double pivot = v->operator[](end);

//Prevent sorting of empty vector

if (end == start)

return;

//partition the vectors

while (left <= right) {

while (v->cmp(left, end))

left++;

while ((v->cmp(end, right)))

right--;

if (left <= right) {

v->swap(left, right);

left++;

right--;

}

}

//recursively sort

if (start < right)

quicksort(v, start, right);

if (left < end)

quicksort(v, left, end);

}

**Function heap\_from\_root with SortableContainer:**

//Generate the heap

void heap\_from\_root(SortableContainer \*v, int i, int n)

{

int child1, child2, largest;

child1 = (2 \* i) + 1;

child2 = child1 + 1;

//check if inbounds and if its bigger than parents

if (child1 <= n && (v->operator[](child1) > v->operator[](i)))

largest = child1;

else//if not, set parent as biggest

largest = i;

//compare child2 with current largest

if (child2 <= n && (v->operator[](child2) > v->operator[](largest)))

largest = child2;

//if the largest isn’t the parent aka it’s a child

if (largest != i)

{

v->swap(i, largest);

}

//we’re at the root node

if (i == 0) {}

if (i > 0)

heap\_from\_root(v, i - 1, n);

}

**Function heap with SortableContainer:**

//Sorts the heap

void heap(SortableContainer \*v)

{

//start at the lastchild’s parent node

int startIndex = floor((v->size() - 2) / 2);

//generate the heap

heap\_from\_root(v, startIndex, v->size() - 1);

int j;

//Remove largest and repeat.

for (j = v->size() - 1; j >= 1; j--)

{

v->swap(0, j);

startIndex = floor((j - 2) / 2);

heap\_from\_root(v, startIndex, j - 1);

}

}

**MVector.h:**

#ifndef MVECTOR\_H // the 'include guard'

#define MVECTOR\_H // see C++ Primer Sec. 2.9.2

#include <vector>

#include "SortableContainer.h"

using namespace std;

// Class that represents a mathematical vector

class MVector : public SortableContainer

{

public:

// constructors

MVector() {}

explicit MVector(int n) : v(n) {}

MVector(int n, double x) : v(n, x) {}

// access element (lvalue)

double &operator[](int index)

{

return v[index];

}

// access element (rvalue)

double operator[](int index) const

{

return v[index];

}

int size() const

{

return v.size();

} // number of elements

void swap(int a, int b);

void initialise\_Random(double xmin, double xmax);

MVector subVector(int start, int end);

bool cmp(int i, int j);

friend ostream& operator<<(ostream& out, const MVector &w) {

out << "(";

for (int i = 0; i < w.size() - 1; i++)

out << w[i] << ", ";

out << w[w.size() - 1] << ")" << endl;

return out;

};

private:

std::vector<double> v;

};

//Member function definitions

void MVector::swap(int a, int b)

{

MVector temp(1);

temp[0] = v[a];

v[a] = v[b];

v[b] = temp[0];

}

void MVector::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);

}

bool MVector::cmp(int i, int j)

{

if (v[i] < v[j])

return true;

else

return false;

}

#endif

**Code that Generates the tables:**

srand(time(NULL));

cout.precision(6);

int n = 5;

double totalTime = 0;

ofstream sortResults;

sortResults.open("heap example.txt");

//if attempt to open file failed, returns 1

if (!sortResults) return 1;

for (int i = 0; i<10; i++)

{

MVector testVector(n);

testVector.initialise\_Random(-100, 100);

double startTime = Timer(); // take a time measurement

bubble(testVector);

double endTime = Timer(); // take another time measurement

// Report duration

sortResults << "Bubble(" << n << ") = " << endl;

sortResults << testVector << endl;

sortResults << "took " << endTime - startTime << "s" << endl;

sortResults << endl;

totalTime += (endTime - startTime);

}

sortResults << "Average Run Time: " << totalTime / 10 << endl;

# Bibliography

*Algorithms - Quicksort*. (n.d.). Retrieved from Khan Academy: https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

*Complexity of Heapsort*. (n.d.). Retrieved from Heap-Comp: https://fmse.info.uaic.ro/getteachingfile/53/heap-comp.pdf/

*Heapsort Analysis and Partitioning*. (1998, March 12). Retrieved from https://www.cs.umd.edu: https://www.cs.umd.edu/~meesh/351/mount/lectures/lect14-heapsort-analysis-part.pdf

*Quicksort*. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Quicksort

Wikipedia. (n.d.). *Bubblesort* . Retrieved from Wikipedia.