Both  Bubble Sort and Quick Sort are well known sorting algorithms in the in the CS community. To be blunt and to the point, Bubble Sort is inefficient compared to Quick Sort. Here’s a generic implementation of Bubble Sort:

var bubbleSort = function(array)

{

var count = array.length — 1;

for(var x = 0; x < array.length; x++)

{

var changes = 0;

for(var m = 0; m < count; m++)

{

var val = array[m];

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

{

array[m] = array[m+1];

array[m+1] = val;

changes++;

}

i++;

}

count--;

if(changes === 0)

{

return array;

}

}

return array;

};

Bubble Sort, on average, has a time complexity of O(n^2), that is, for 2 elements in the array, there are 4 operations that happen, for 3, 9 operations, for 4, 16 operations, and so on. For each element, the algorithm iterates through the whole array. This becomes a problem when you have thousands of elements you’re trying to sort.

Quick Sort, on the other hand, has an average time complexity of O(n log n), meaning that it is less efficient than a linear algorithm, but each element added to the array is less expensive than the last. This property is invaluable compared to a quadratic algorithm. Here’s an example of a Quick Sort implementation:

var quickSort = function(a)

{

if (a.length == 0) return [];

var left = [], right = [], pivot = a[0];

for (var i = 1; i < a.length; i++)

{

a[i] < pivot ? left.push(a[i]) : right.push(a[i]);

}

return quickSort(left).concat(pivot, quickSort(right));

}

Quick Sort recursively (or iteratively, based on implementation) splits the array, and subsequent parts, into left and right arrays, based on a pivot value. Worst case scenario, your pivot point is either the greatest or smallest element in the array, in which case, your time complexity increases to O(n^2), but if you choose a random pivot point, this shouldn’t be something you have to worry about too much.

Bubble sort is commonly used by beginners in CS because it is logically easier to understand. After working with Tree data structures, Quick Sort’s binary sort method becomes more clear.

Given that average case for Bubble Sort is the worst case for Quick Sort, it is safe to say that Quick Sort is the superior sorting algorithm. For short arrays (under 1,000 elements), the benefits of Quick Sort are minimal, and might be outweighed by it’s complexity, if the goal is readability. But for production level projects, Quick Sort is the way to go. Some might argue that Quick Sort’s name gives it away.

Comparing different sorting algorithms for time performance has always been amusing. It has been done tons of time. But you should try it out for yourself. Apart from fun, this comparison is very useful in real life. Companies and organizations need to sort and search data all the time. Thus it makes sense to compare and find the best suitable sorting method.

Code for Comparison between Bubble sort & Quick sort :

#include <iostream>

#include <fstream>

#include <cstdlib>

#include <ctime>

using namespace std;

long length = 100000;

const long max\_length = 400000;

int list[max\_length];

void read()

{

ifstream fin("input8137.txt", ios::binary);

for (long i = 0; i < length; i++)

{

fin.read((char\*)&list[i], sizeof(int));

}

fin.close();

}

void bubbleSort()

{

int temp;

for(long i = 0; i < length; i++)

{

for(long j = 0; j < length-i-1; j++)

{

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

{

temp = list[j];

list[j] = list[j+1];

list[j+1] = temp;

}

}

}

}

long partition(long left, long right)

{

int pivot\_element = list[left];

int lb = left, ub = right;

int temp;

while (left < right)

{

while(list[left] <= pivot\_element)

left++;

while(list[right] > pivot\_element)

right--;

if (left < right)

{

temp = list[left];

list[left] = list[right];

list[right] = temp;

}

}

list[lb] = list[right];

list[right] = pivot\_element;

return right;

}

void quickSort(long left, long right)

{

if (left < right)

{

long pivot = partition(left, right);

quickSort(left, pivot-1);

quickSort(pivot+1, right);

}

}

int main()

{

double t1, t2;

for (length = 100000; length <= max\_length; )

{

cout << "\nLength\t: " << length << '\n';

read();

t1 = clock();

bubbleSort();

t2 = clock();

cout << "Bubble Sort\t: " << (t2 - t1)/CLK\_TCK << " sec\n";

read();

t1 = clock();

quickSort(0, length - 1);

t2 = clock();

cout << "Quick Sort\t: " << (t2 - t1)/CLK\_TCK << " sec\n";

switch (length)

{

case 100000 :

length = 200000;

break;

case 200000 :

length = 400000;

break;

case 400000 :

length = 400001;

break;

}

}

return 0;

}

Output of given code :

Length : 100000

Bubble Sort : 62.031 sec

Quick Sort : 0.046 sec

Length : 200000

Bubble Sort : 281.267 sec

Quick Sort : 0.234 sec

Length : 400000

Bubble Sort : 1010.52 sec

Quick Sort : 0.915 sec

Conclusion:

I would conclude this, that

* Bubble Sort is not suitable in any circumstance. It is an O(n2) algorithm with a large constant. In simple words, time required to perform bubble sort on ‘n’ numbers increases as square of ‘n’. Thus it is quite slow.
* Quick Sort amazed me. Well it is an O(n\*log(n)) algorithm on an average case and an O(n2) algorithm in the worst case scenario. (Quick sort’s worst case occurs when the numbers are already sorted!!).