An example of bubble sort. Starting from the beginning of the list, compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one). After each [iteration](https://en.wikipedia.org/wiki/Iteration), one less element (the last one) is needed to be compared until there are no more elements left to be compared.

Q: Tiến hành sắp xếp trong máy tính bằng thuật toán sắp xếp nổi bọt (bubble sort), thì 1000 dữ liệu mất 1 giây. Nếu sắp xếp 5000 dữ liệu tương tự thì phải mất bao nhiêu lâu?

A: Thời gian so sánh trung bình của bubble sort là O(n2). Trong đó n là số phần tử, số data. Chữ O ở đây là đại diện cho 1 phép toán phức tạp nào đó.

O(1000^2) = 1 giây

O(5000^2) = ? giây

? = [(5000\*5000)\*1] / (1000\*1000) = (25\*1000\*1000) / (1000\*1000) = 25 giây

Thực hiện chuyển đổi dữ liệu là swap

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

First Pass

( **5** **1** 4 2 8 ) \to( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) \to( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) \to( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) \to( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass

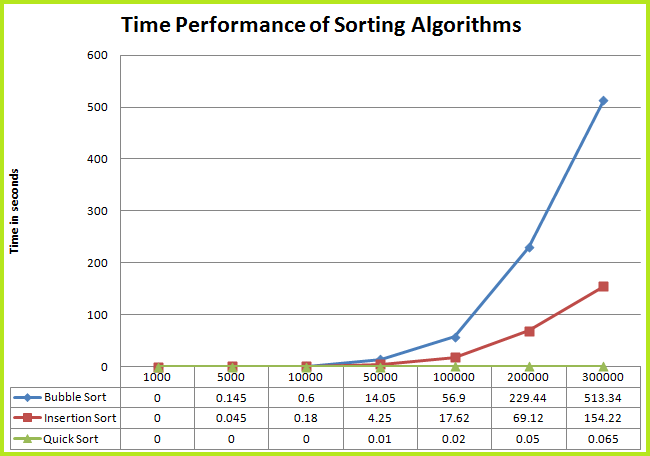
( **1** **4** 2 5 8 ) \to( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) \to( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) \to( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) \to( 1 2 4 **5** **8** )  
Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass

( **1** **2** 4 5 8 ) \to( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) \to( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) \to( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) \to( 1 2 4 **5** **8** )

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

First let me show you the result in a neat graph (courtesy MS Excel charts), displaying **time** (on y-axis) v/s **list size** (on x-axis).

[](https://vinayakgarg.files.wordpress.com/2011/10/comparison.png)

If you are done analysing the graph, please note following points

* The x-axis represents the number of elements sorted.
* The x-axis is not linear (nor logarithmic). Its random, you see 1,000 then 5,000 then 10,000 and so on up to 3,00,000. The last three entries are linear however, and shows the highlight feature of the graph. The difference in the three algorithms performance.
* Quick Sort seems to not want to let go off the axis. It is not a mistake, it is a reality. A sweet reality.
* Bubble sort’s curve would make a dream come true graph for a company’s profit. Winking smile

Here I reproduce the result in tabular form –

|  |  |  |  |
| --- | --- | --- | --- |
| Length | Bubble Sort | Insertion Sort | Quick Sort |
| 1000 | 0 | 0 | 0 |
| 5000 | 0.145 | 0.045 | 0 |
| 10000 | 0.6 | 0.18 | 0 |
| 50000 | 14.05 | 4.25 | 0.01 |
| 100000 | 56.9 | 17.62 | 0.02 |
| 200000 | 229.44 | 69.12 | 0.05 |
| 300000 | 513.34 | 154.22 | 0.065 |

### Variations in numbers (time recorded)

Consider Insertion Sort’s time taken for 5000 integers, 0.045 seconds. This is an average value. Due to other processes going on at the same time as comparison, the recorded time varies during each run. It also varies from computer to computer (mine one is a decent one though, Intel i3 with 4 GB of RAM). Like in one execution, above time was recorded as 0.04 sec and in other it was 0.05 sec. Same goes for all the time values. In effect I have averaged the time taken in the 4 executions. For greater accuracy keep the number of samples to a minimum 10 (although I leave it to you). And also the code should be executed on a variety of computers and Operating Systems (again I leave that for you). However the differences are so pronounced that a variation of 5-10% does not affect the end result.

### The big question?

You must wonder why I chose the above three algorithms for comparison. Why not merge sort (arguably the fastest sorting algorithm), or heap sort? And pray, why include Bubble Sort at all??

For one the beginners should know that their favourite sorting (Bubble sort), is umm, well, pathetic and un-acceptable. Second, that quick sort is indeed quick. And I had to take some medium performing algorithm, so Insertion Sort was chosen. All three of them are quite popular and easy to understand.

Give me the code

Yeah sure, here it is:

#include <iostream>

#include <fstream>

#include <cstdlib>

#include <ctime>

using namespace std;

long length = 1000;

const long max\_length = 300000;

int list[max\_length];

void read()

{

ifstream fin("random.dat", 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;

}

}

}

}

void insertionSort()

{

int temp;

for(long i = 1; i < length; i++)

{

temp = list[i];

long j;

for(j = i-1; j >= 0 && list[j] > temp; j--)

{

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

}

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 = 1000; 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();

insertionSort();

t2 = clock();

cout << "Insertion 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 1000 :

length = 5000;

break;

case 5000 :

length = 10000;

break;

case 10000 :

length = 50000;

break;

case 50000 :

length = 100000;

break;

case 100000 :

length = 200000;

break;

case 200000 :

length = 300000;

break;

case 300000 :

length = 300001;

break;

}

}

return 0;

}

I encourage you to copy the code (modify it to suit your needs), compile it with your choice of compiler and execute it. And share the results with us. I will be glad to showcase here what you found (and how fast your computer runs Smile)

IMPORTANT : The above code requires a file ‘**random.dat**‘ which contains 3,00,000 integers stored randomly. This file is read each time before a sorting is performed. You can download it here <http://www.box.net/shared/1n3r3hedv86351iun6f5> or here [https://skydrive.live.com/?cid=a976bcd81e2434ba&sc=documents&id=A976BCD81E2434BA%21114#](https://skydrive.live.com/?cid=a976bcd81e2434ba&sc=documents&id=A976BCD81E2434BA%21114) The file is 1.14MB sized.  
Alternatively you can generate ‘**random.dat**‘ file through this code. This way you can generate a bigger file for more numbers.

#include <fstream>

#include <cstdlib>

#include <ctime>

using namespace std;

const size\_t length = 300000;

int main()

{

ofstream fout("random.dat", ios::binary);

srand(time(NULL));

int r;

for (size\_t i = 0; i < length; i++)

{

r = rand();

fout.write((char\*)&r, sizeof(r));

}

fout.close();

return 0;

}

Also you can download both the codes, there executables and random.dat in this RAR file(1.01 MB) <http://www.box.net/shared/31ofn1v53qrdm0y2cdce> or here [https://skydrive.live.com/?cid=a976bcd81e2434ba&sc=documents&uc=1&id=A976BCD81E2434BA%21116#](https://skydrive.live.com/?cid=a976bcd81e2434ba&sc=documents&uc=1&id=A976BCD81E2434BA%21116)

WARNING : The total execution time can exceed 20 minutes (or even more) on slow processors. It may seem that program is hung after sorting 50,000 numbers, but it is not so. The screen stops responding because, the bubble sort starts taking longer times (in several minutes). So nothing to worry, just watch a video or two on vimeo (or youtube).

### 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.
* Insertion Sort is suitable for small files, but again it is an O(n2) algorithm, but with a small constant. Also note that it works best when the file(/numbers) is already almost sorted.
* Quick Sort amazed me. Didn’t you get amazed?! 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!!) The graph speaks it all. You need this algorithm when the list is large and time is premium.