CAB301 Algorithms and Complexity

Nathan Perkins

April 21, 2016

Contents

1 Executive Summary

2 Bubble Sort

2.1 Algorithm

The general Bubble Sort algorithm works as follows. For every element in the array, successively compare each pair of adjacent elements, up until the total array size minus the current index, swapping each checked pair based on the comparison. For ascending sort, the comparison is if the previous element is larger than the next, and for descending sort the comparison is if the next element is larger than the previous. The reason that not all adjacent pairs are checked each iteration is due to the fact that for each iteration, the current highest element will shift to it's appropriate position in the array. For every iteration over the array, n, the nth element from the end of the listed is correctly indexed and therefore no longer needs to be checked with further iterations. Appendix A lists the pseudo code example for the algorithm.

2.2 Average Case Efficiency

Due to the nature of the algorithm, iterating over the entire loop successively, whilst check and swapping each adjacent pair, the algorithm is inefficient when compared to other methods of sorting. Bubble Sort has a computational efficiency on the order of $\mathcal{O}(n^2)$, with a more specific computational efficiency of $\frac{n^2}{2} + \frac{n}{2}$. [1] In contrast, more efficient algorithms like Heap Sort or Quick Sort have efficiencies on the order of $\mathcal{O}(n \log n)$, which grows on a much slower scale than Bubble Sort. [2]

3 Better Bubble Sort

3.1 Algorithm

The basic Bubble Sort algorithm can be improved by introducing a swap flag. Without the swap flag, the algorithm will continue to compare elements of the array, regardless of whether the array has been sorted. A swap flag, which drives the main loop will ensure that the algorithm exits prematurely should the array elements reach a sorted state. Appendix A lists the better bubble sort algorithm using pseudo code.

3.2 Basic Operation

The operation chosen, to analyse the efficiency of the algorithm is the number of individual swap operations. This aggregate operation defines the core design of the Bubble Sort algorithm, with it being common to both designs, basic and improved.

3.3 Average Case Efficiency

The efficiency of the Better Bubble Sort algorithm is very similar to the original algorithms efficiency. It is of the same order as the original algorithm, as the fundamental design of it's operation hasn't changed. However, the addition of the swap flag allows the algorithm to exit prematurely if it has completely sorted the array before the full algorithm has completed. Due to this check flag, it is very hard to quantify the improvements that this addition makes using theoretical analysis alone, as it is highly dependant on the ordering of the array given to the algorithm. Since the overall design of the algorithm hasn't significantly changed, it is expected that the general order of growth will be $\mathcal{O}(n^2)$, whilst it should be on average slightly faster than the original.

Constants are ignored when comparing efficiency and therefore two different algorithms can have the same order of growth, but still perform differently. The order of growth is calculated by examining the algorithm, which can be seen in Appendix A, and estimating the number of iterations required to reach completion. In Bubble sort, there are 2 defining loops which contribute to its efficiency. The outermost loop must iterate a number of times equal to the array size, whilst the innermost loop will similarly iterate a number of times equal to the array size minus the current index of the outer loop. Since the loops are embedded, the general order of magnitude is taken by multiplying both together, and only considering the highest power, results in the above efficiency.

4 Methodology

4.1 Computing Environment

For experimental analysis, the computing environment can play a pivotal role in the specific results and has therefore been documented.

1. Operating System: The operating system used for the construction, data generation and analysis of report is Linux, specifically Ubuntu 15.10. This was chosen as it includes easy to use, pre-installed packages that aid in the programming of C++ and python which was also used. All code was then checked to conform with Microsoft Windows for veracity of design.

2. Algorithm: For creation of the algorithm, C++ was used. C++ is an efficient, and fast language that gives good freedom over design and memory, whilst still maintaining high level functionality. The Eclipse cross platform IDE was chosen to program the algorithm in, which gave access to debugging tools and automated compiling. For veracity, this was later ported to the code::blocks IDE and tested.

3. Data Analysis

   (a) Data types: Array size, number of iterations to sort the array, as well as time taken in seconds to complete are stored.

(b) Storage: CSV file format was used extensively to store the data directly via the C++ executable. CSV, or comma separated variables is an easy to use file format that separates all variables through the use of commas, this makes it perfect to easily get file formatting and output with C++. Python has in built functionality to interpret CSV files and was also a good choice in that regards, allowing the easy import of the data for plotting.

(c) Analysis: Python was used to interpret the data taken from CSV into graphs using the matplotlib graphing library. Python is a cross platform, high level scripting language with lots of in built scientific analysis features and is therefore a good choice to evaluate the data from any location or machine. matplotlib is a free library that easily plots data.

4. Report: To build the report and everything included, LaTeX was used. LaTex allows good control over document flow and it's primary purpose, the easy inclusion of mathematical formulae and equations fits perfectly with the overall design of the report.

4.2 Test Data Generation

To test the algorithm, it is important to note the types of arrays used as an input. Arrays sorted in specific manners can skew results, and lead to different performance than intended. As the main test of the report is on the average case, the generation of randomised arrays was of utmost importance, however, for testing, best case and worst case array generation was also performed.

1. Random arrays: Initialising the random seed based on the current system time, then generate each element as a random number from 1-100. With each element being a random number with no association to the previous or next element, a truly randomised list is created.

2. Ordered arrays: Initialising the random seed based on the current system time, setting the first element to be 1 and then proceeding to generate a random number between 0-9 and adding it to the previous array element and setting the next element equal to the result ensures that a strictly ascending array is generated.

3. Reversed arrays: Initialising the random seed based on the current system time, setting the last element to be 1 and then proceeding to generate a random number between 0-9 and adding it to the next array element and setting the previous element equal to the result ensure that a strictly descending array is generated.

4. Data Generation: The data for the overall testing is generated sequentially with increasing array sizes, repeated a number of times and then the average for that array size is then checked for validity and saved to

the CSV. Data can then be presented as a continuous graph. Number of repeated operations was chosen as 100, and the array sizes saved are between $2 \leq n < 2000$

5 Implementation

5.1 Program

The bubble sort algorithm is implemented in c++, and can be seen in Appendix B. As mentioned in the methodology section, randomised arrays were created and then sorted, counting the number of swap operations. Appendix B lists the code used to generate an array, as well as the functional code to generate an ascending order of increasing size(n) arrays. The function srand was initialised in main, instead of the GenerateArray function, so as to avoid problems with same seed timing issues.

The main function instantiates a number of required variables, including file names, steps, duration, maximum array size, number of times to repeat. It then proceeds to generate random arrays up to the maximum size, and repeating to get the average of each array size, then returning that average for both number of steps, representing computational operations, and the time taken for each sorting. This can be seen in Appendix B, the main function.

5.2 Testing

For testing purposes, a helper function was used throughout the whole development process, as well as actively being used in the data generation in the main function. IsSorted, as seen in Appendix B takes an array and checks if it's correctly sorted in ascending order.

For validity and checking purposes, reversed arrays and ordered arrays are also generated and compared against their theoretical results, these functions are also seen in Appendix B.

6 Experimental Results

6.1 Operational Efficiency

An addition has been made to the algorithm through the usage of num_steps variable which keep track of the number of swaps made and is incremented on the inner most loop, corresponding to each time an array swap has occurred. This can be seen in the comments of the code in Appendix B. The addition of the variable doesn't affect the overall performance in regards to growth of operational complexity. It has only a trivial affect on the duration taken for each sorting as it's only a single instruction addition to the algorithm.

Comparing against the average case efficiency of the base algorithm, it can be seen in Appendix C that the swap flag version of the algorithm fares significantly better in computational complexity. As mentioned in methodology, data was generated for each array size between 2 to 2000, repeated 100 times each and the average was taken. The average case of the efficiency was computed by creating randomly sorted arrays, as mentioned in the Implementation and methodology sections. The measured computational steps grows on an order similar to the average case for un-optimized bubble sort, however, it grows at a slower rate in that order class. This is expected and in line with theoretical results. The addition of the swap flag allows the algorithm to exit early if its completed sorting, and thus it's expected that the efficiency be less than or in extreme cases, equal to the efficiency of the un-optimised algorithm.

To compare further against the un-optimised algorithm, the very worst case was computed for the optimised algorithm. The worst case for the optimised algorithm is when the swap flag gets no opportunity to exit prematurely. This will be when the array is completely reversed. Data was again generated as per the methodology section, with incremental array sizes from 2 to 2000, and the average of 100 cases per array size was taken. Results can be seen in Appendix C, The Worst case for the optimized version is the same number of computations for the un-optimized average case.
You must present your experimental results as a graph. NB: You must state clearly how many data points contribute to the line(s) on the graph and what each data point represents. If possible, you should use a graph drawing tool that displays each data point as a distinct symbol.
You must state whether or not the experimental results matched the predicted number of operations. If they do not match then you must offer some explanation for the discrepancy. Normally we would expect that counting basic operations produces results that closely match the theoretical predictions, but it is possible that there is some peculiarity of your experimental set-up that skews the results, or even that the theoretical predictions are wrong.

6.2 Time Efficiency

Your report must explain clearly how you measured execution times, e.g., by showing the relevant test program. (Alternatively, you may even choose to time your program with a stopwatch, although this is unlikely to produce accurate results.) It is often the case that small program fragments execute too quickly to time accurately. Therefore, you may need to time a large number of identical tests and divide the total time by the number of tests to get useful results.
You must perform sufficient experiments to produce a clear trend in the outcomes. Your report must make clear how you produced test data (as per the discussion above on counting basic operations).
You must present your experimental results as a graph. NB: You must state clearly how many data points contribute to the results on the graph and what each data point represents. If possible, you should use a graph drawing tool

that displays each data point as a distinct symbol.
You must state whether or not the experimental results matched the predicted order of growth. It is possible that your measured execution times may not match the prediction due to factors other than the algorithms behaviour, and you should point this out if this is the case in your experiments. For instance, an algorithm with an anticipated linear growth may produce a slightly convex scatterplot due to operating system and memory management overheads on your computer that are not allowed for in the theoretical analysis. (However, a concave or totally random scatterplot is more likely to be due to errors in your experimental methodology in this case!)

7 Appendix A - Algorithm

The following is a pseudo code example of the non-optimized bubble sort algorithm.

Figure 1: Basic Bubble Sort

```
func bubblesort( var a as array )
  for i from 1 to length(a)
    for j from 0 to length(a) - i
      if a[j] > a[j + 1]
        swap( a[j], a[j + 1] )
end func
```

The improved version of the Algorithm, listed below, adds in a swap flag which enables the function to exit prematurely should no swaps be made over an entire iteration of the outer loop.

Figure 2: Improved Bubble Sort

```
func bubblesort( var a as array )
  bool swapped = true
  count = length(a)
  while (swapped)
    swapped = false
    for j from 0 to length(a) - count
      if a[j] > a[j + 1]
        swap( a[j], a[j + 1] )
        swapped = true
    count = count - 1
end func
```

8 Appendix B - C++ Implementation

Figure 3: Better Bubble Sort C++ Implementation

```
int BetterBubbleSort(int array[], int size){
  int num_steps = 0; // This tracks the operation count
  int count = size-1;
  bool sflag = true;
  while (sflag){
    sflag = false;
    for (int j = 0; j<=count-1;j++){
      if (array[j+1] < array[j]){
        //Swap the 2 array elements
        int temp = array[j];
        array[j] = array[j+1];
        array[j+1] = temp;
        sflag = true;
        num_steps++; //Increment the operation counter
      }
    }
    count--;
  }
return num_steps; //Return the operation count
}
```

Figure 4: Random Array Generation

```
  int main() {

    srand(time(NULL));
    ...
  }
  ...
  void GenerateArray(int array[], int size){
    for(int i = 0; i<size;i++){
      int temp = rand()%100+1;
      array[i] = temp;
    }
  }
```

Seen below is the Main function for generating data. An initial size, and repeat variables are instantiated, as well as variables for checking duration of sorting function, where to save data and the number of steps taken. For each size of the array, i, it repeats the process a number of times based on the repeat variable. It then generates a random array, starts a timing function, sorts the array and then returns the time taken and number of steps. It then checks if the array is sorted and saves the average of the full data, after being repeated a number of

times. It then repeats the exact sequence for reversed and ordered arrays for checking and validity purposes, not included here for space.

Figure 5: Main Function

```
//============================================================================
// Name        : BetterBubbleSort.cpp
// Author      : Nathan Perkins
// Version     :
// Copyright   : No Copyright
// Description : Hello World in C++, Ansi-style
//============================================================================

#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <fstream>

using namespace std;

int BetterBubbleSort(int array[], int size);
bool IsSorted(int array[],int size);
void SortedPrint(int array[], int size);
void PrintArray(int array[], int size);
void GenerateArray(int array[], int size);
void GenerateOrderedArray(int array[],int size);
void GenerateReversedArray(int array[], int size);
bool SaveData(int num, int n, double time, char filename[]);

int main() {
  srand(time(NULL));
  int size = 2000;
  int repeat = 100;
  int steps;
  clock_t start;
  double duration;
  int i = 2;
  char filename[] = "BubbleSort.csv";
  char filename1[] = "BubbleSortOrdered.csv";
  char filename2[] = "BubbleSortReversed.csv";

  while (i<size){
    steps = 0;
    duration = 0.0;
    for(int j = 0;j<repeat;j++){
      int array[i];
      GenerateArray(array,i);
      start=clock();
      steps += BetterBubbleSort(array,i);
      duration += ( clock() - start)/(double) CLOCKS_PER_SEC;
      if(IsSorted(array,i)){
        if(j==0){
          cout<<"Iteration "<< i << " sorted"<< endl;
        }
      }
    }
    steps = steps/repeat;
    duration = duration/(double)repeat;
    if(!SaveData(steps,i,duration, filename)){
      cout<<"Writing to file failed:"<<endl;
      cout<<"Number: "<<i<<endl;
      cout<<"Steps:  "<<steps<<endl;
      cout<<"Time:   "<<duration<<endl;
      break;
    }
    i++;
  }
  cout<<"Finished Writing Random to file"<<endl;
  ...
}
```

Figure 6: Checking if Array is sorted

```
bool IsSorted(int array[], int size){
  for(int i=1;i<size;i++){
    if(array[i]<array[i-1]){
      return false;
    }
  }
  return true;
}
```

Figure 7: Reversed and Ordered Array Generation

```
void GenerateOrderedArray(int array[], int size){
  array[0] = 1;
  for(int i = 1; i<size;i++){
    int temp = rand()%10+array[i-1];
    array[i] = temp;
  }
}
void GenerateReversedArray(int array[], int size){
  array[size-1] = 1;
  for(int i = size-2; i>=0;i--){
    int temp = rand()%10+array[i+1];
    array[i] = temp;
  }
}
```

9 Appendix C - Experimental Results

As array size (n) increases, as seen below, it can be seen that the measured computational steps increases at a slower rate than $\frac{n^2}{2}$, denoted by the light blue line.

Figure 8: Bubblesort Operation Comparison: Expected vs Actual

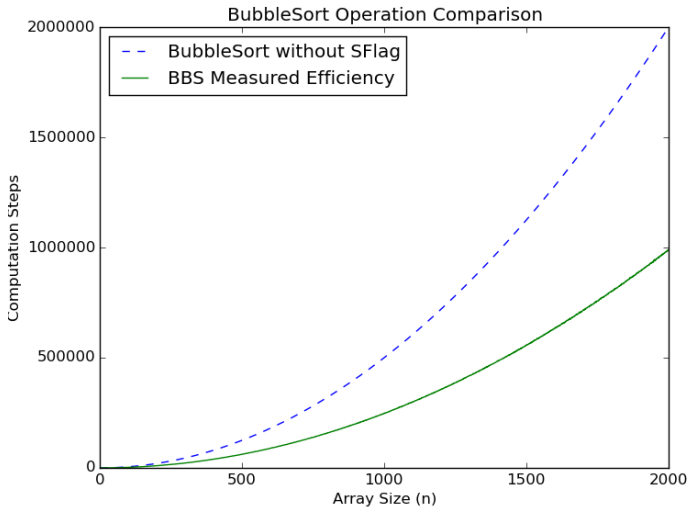
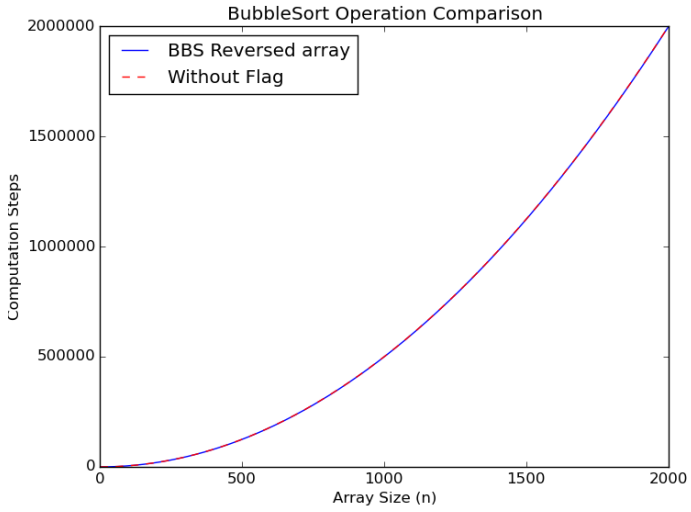


Figure 9: Bubblesort Reversed Array against average case



References

[1] W. Min, "Analysis on bubble sort algorithm optimization," in *Information Technology and Applications (IFITA), 2010 International Forum on*, vol. 1.

IEEE, 2010, pp. 208–211.

[2] R. Schaffer and R. Sedgewick, "The analysis of heapsort," *Journal of Algorithms*, vol. 15, no. 1, pp. 76–100, 1993.