

Sorting Algorithms

Arden Rasmussen

September 23, 2017

Contents

1	Comparison Sorts	2
1.1	Bubble Sort	3
1.2	Bubble Sort	6
1.3	Bubble Sort	9
1.4	Bubble Sort	12
1.5	Bubble Sort	15
1.6	Bubble Sort	18
	Appendices	21
A	Algorithms	21
A.1	Bubble Sort	21

Abstract

Sorting algorithms are used in many aspects of the modern computational world, and vital for data sciences. However, with the vast variety of different sorting algorithms, it is important to be able to utilize the algorithm that is best suited for a data set, instead of applying a general solution that will run at a sub optimal rate. This paper aims to classify a wide selection of sorting algorithms for the different data sets that they are most optimally used in conjunction with.

1 Comparison Sorts

Comparison sorting algorithms, are algorithms that directly compare the entries of the set of items to be sorted. This means that each item is checked against one another and depending on the check, these items may be swapped to further the order of the array of items, until the array is completely sorted. These sorting algorithms are some of the most common, because it is fairly simple to implement, and to theoretically think of.

This means that there is a large number of comparison sorts, that are commonly used in the world of computer science. Some of the most common are Bubble sort, Quicksort or Merge sort. One aspect of comparison sort algorithms is that they tend to use less memory, than that of non comparison sorting algorithms, because the comparison sorts tend to swap the elements of the list in place, instead of creating new lists, and copying. However, they tend to be limited because they are directly comparing each element to one another.

Contents

1.1	Bubble Sort	3
1.2	Bubble Sort	6
1.3	Bubble Sort	9
1.4	Bubble Sort	12
1.5	Bubble Sort	15
1.6	Bubble Sort	18

1.1 Bubble Sort

Best-Case Performance	$O(n)$
Average Performance	$O(n^2)$
Worst-Case Performance	$O(n^2)$
Worst-Case Space Complexity	$O(1)$

1.1.1 Introduction

Bubble sort is an algorithm that goes item by item, sorting pairs of two adjacent items at a time. This is where the name *Bubble sort* originates from, due to the fact that the algorithm creates bubbles around two adjacent items at a time and only considers the order of these two. This is one of simplest comparison sorting algorithms, as it can be implemented in a few number of lines of code, and is simple to conceptualize. The algorithm runs through the list of items swapping pairs as needed, until no more swaps were made on a single pass, which would mean that each item is in its position.

The worst case complexity of bubble sort is $O(n^2)$, because each item can only be moved one position in a pass, meaning that for each item that is out of position must be slowly moved into position. For even relatively small lists of data, this can take times which are far too large to be useful beyond a demonstration. There is one case where bubble sort can provide an advantage over other sorting algorithms. Because the entire process of the sorting, is checking if the items are in a sorted order, if the list is already in a sorted order, Bubble sort will make a single pass testing if everything is in a sorted order. After it finishes the single pass it will determine that the list is already sorted, and end.

Bubble sort has two elements that cause sorting to be extremely slowed. These are “turtles”, and “rabbits”.

“Rabbits” are elements that are at the beginning of the list and belong at the end of the list. Because bubble sort moves from the first to the last element on a repeating order, these “rabbits” will quickly move from the front to the

end, because everytime they are compared to a another value, they are found to be greater than that value, and are swapped, until they reach their final position. They are called “rabbits” due to the fact that they will move to their final destination on the first pass, unless a “faster rabbit” is found. Below is an example of a “rabbit”, where 9 is the “rabbit”.

```

925580
925580
295580
259580
255980
255890
255809

```

“Turtles” are items that are near the end of the list, that belong at the beginning. These are called “turtles” due to the fact that they cause the algorithm to run for much longer, and cannot move quickly to their final destination. The “turtle” can only move once per pass, as they are only moved for a single comparison, before the bubble moves on to the next adjacent pair. These “turtles” are what commonly cause the worst case performance that is found in Bubble sort. As they can strictly only move one position for each pass, they can at most require $n - 1$ passes to move to their final position, where n is the number of terms in the list. Below is an example of a “turtle”, where 0 is the “turtle”. Each row is a new pass of the algorithm, where the bubble has passed through all the elements, similar to that demonstrated in the example for

“rabbits”, essentially there are 5 steps between each step shown in the following example.

```

          925580
1st...255890
2ed...255809
3ed...255089
4th...250589
5th...205589
          025589

```

1.1.2 Algorithm

Algorithm 1.1.A Bubble Sort

```

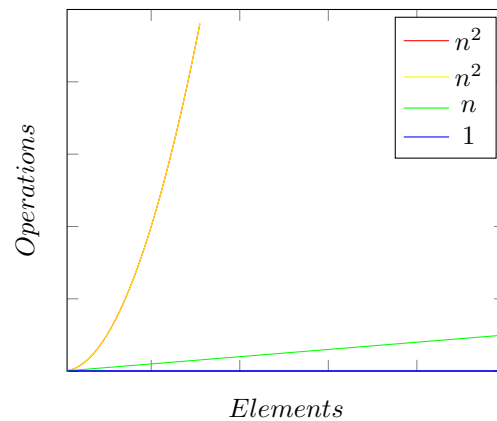
function BUBBLE(a)
  length  $\leftarrow$  length(a)
  while length  $\neq$  0 do
    newLength  $\leftarrow$  0
    for i = 1 to length - 1 do
      if a[i - 1] > a[i] then
        Swap(a[i - 1], a[i])
        newLength  $\leftarrow$  i
      end if
    end for
    length  $\leftarrow$  newLength
  end while
  return a
end function

```

This is a simple pseudocode algorithm for a bubble sort implementation. The algorithm begins by getting a value for the length of the array (*length*). Then the algorithm runs while this value is not equal to zero. Every time the algorithm begins a new pass through the array, it creates a new length value (*newLength*). Then the algorithm runs through the values of the array from the first value to the value at *length*. The *newLength* value is set to the position of the last swap. This is due to the fact that beyond that position, there where no new swaps, so

the data must already be sorted. This practice is to remove some additional time, as there is no reason for the algorithm to continue to check the already sorted data. The *length* value is set to the value of *newLength*, then the process begins. As the value of *length* becomes shorter, the algorithm is getting closer to completion. When the value of *length* becomes 0, this means that there were no more swaps in the list, and that must imply that all the values have been sorted correctly.

1.1.3 Analysis



This is a plot for the theoretical run times that Bubble sort should provide based on the number of elements provided. As it can be seen the best and average cases (which are the same) grow extremely rapidly. The the rare best case preformance (which only will occur if a single pass is necessary, e.g. the array is already sorted) is much lower. The memory usage is constant at 1 as the sorting algorithm does not need to store or make copies of the data, simply to view and make swaps to the data. The exponential nature of the average case time complexity, means that this algorithm will rapidly grow to an unsuable length of time required to run. As it can clearly be seen in the implementation data below.

1.1.4 Conclusion

The bubble sort algorithm, is extremely inefficient, and should only be used for demonstrational purposes, or if the data is near the correct order. Because the algorithm is effective at checking the set of data, and can simply move a

position one or two spots with only a maximum of two or three passes required. However, anything beyond this causes the algorithm to grow exponentially in time complexity.

Implementation examples for Bubble sort can be found in A.1.

1.2 Bubble Sort

Best-Case Performance	$O(n)$
Average Performance	$O(n^2)$
Worst-Case Performance	$O(n^2)$
Worst-Case Space Complexity	$O(1)$

1.2.1 Introduction

Bubble sort is an algorithm that goes item by item, sorting pairs of two adjacent items at a time. This is where the name *Bubble sort* originates from, due to the fact that the algorithm creates bubbles around two adjacent items at a time and only considers the order of these two. This is one of simplest comparison sorting algorithms, as it can be implemented in a few number of lines of code, and is simple to conceptualize. The algorithm runs through the list of items swapping pairs as needed, until no more swaps were made on a single pass, which would mean that each item is in its position.

The worst case complexity of bubble sort is $O(n^2)$, because each item can only be moved one position in a pass, meaning that for each item that is out of position must be slowly moved into position. For even relatively small lists of data, this can take times which are far too large to be useful beyond a demonstration. There is one case where bubble sort can provide an advantage over other sorting algorithms. Because the entire process of the sorting, is checking if the items are in a sorted order, if the list is already in a sorted order, Bubble sort will make a single pass testing if everything is in a sorted order. After it finishes the single pass it will determine that the list is already sorted, and end.

Bubble sort has two elements that cause sorting to be extremely slowed. These are “turtles”, and “rabbits”.

“Rabbits” are elements that are at the beginning of the list and belong at the end of the list. Because bubble sort moves from the first to the last element on a repeating order, these “rabbits” will quickly move from the front to the

end, because everytime they are compared to a another value, they are found to be greater than that value, and are swapped, until they reach their final position. They are called “rabbits” due to the fact that they will move to their final destination on the first pass, unless a “faster rabbit” is found. Below is an example of a “rabbit”, where 9 is the “rabbit”.

```

925580
925580
295580
259580
255980
255890
255809

```

“Turtles” are items that are near the end of the list, that belong at the beginning. These are called “turtles” due to the fact that they cause the algorithm to run for much longer, and cannot move quickly to their final destination. The “turtle” can only move once per pass, as they are only moved for a single comparison, before the bubble moves on to the next adjacent pair. These “turtles” are what commonly cause the worst case performance that is found in Bubble sort. As they can strictly only move one position for each pass, they can at most require $n - 1$ passes to move to their final position, where n is the number of terms in the list. Below is an example of a “turtle”, where 0 is the “turtle”. Each row is a new pass of the algorithm, where the bubble has passed through all the elements, similar to that demonstrated in the example for

“rabbits”, essentially there are 5 steps between each step shown in the following example.

```

          925580
1st...255890
2ed...255809
3ed...255089
4th...250589
5th...205589
          025589

```

1.2.2 Algorithm

Algorithm 1.2.B Bubble Sort

```

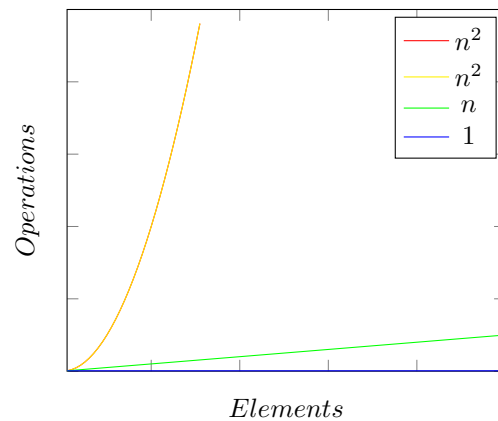
function BUBBLE(a)
  length  $\leftarrow$  length(a)
  while length  $\neq$  0 do
    newLength  $\leftarrow$  0
    for i = 1 to length - 1 do
      if a[i - 1] > a[i] then
        Swap(a[i - 1], a[i])
        newLength  $\leftarrow$  i
      end if
    end for
    length  $\leftarrow$  newLength
  end while
  return a
end function

```

This is a simple pseudocode algorithm for a bubble sort implementation. The algorithm begins by getting a value for the length of the array (*length*). Then the algorithm runs while this value is not equal to zero. Every time the algorithm begins a new pass through the array, it creates a new length value (*newLength*). Then the algorithm runs through the values of the array from the first value to the value at *length*. The *newLength* value is set to the position of the last swap. This is due to the fact that beyond that position, there where no new swaps, so

the data must already be sorted. This practice is to remove some additional time, as there is no reason for the algorithm to continue to check the already sorted data. The *length* value is set to the value of *newLength*, then the process begins. As the value of *length* becomes shorter, the algorithm is getting closer to completion. When the value of *length* becomes 0, this means that there were no more swaps in the list, and that must imply that all the values have been sorted correctly.

1.2.3 Analysis



This is a plot for the theoretical run times that Bubble sort should provide based on the number of elements provided. As it can be seen the best and average cases (which are the same) grow extremely rapidly. The the rare best case preformance (which only will occure if a single pass is neccasary, e.g. the array is already sorted) is much lower. The memory usage is constant at 1 as the sorting algorithm does not need to store or make copies of the data, simply to view and make swaps to the data. The exponential nature of the average case time complexity, means that this algorithm will rapidly grow to an unsuable length of time required to run. As it can clearly be seen in the implementation data below.

1.2.4 Conclusion

The bubble sort algorithm, is extremely inefficient, and should only be used for demonstrational purposes, or if the data is near the correct order. Because the algorithm is effective at checking the set of data, and can simply move a

position one or two spots with only a maximum of two or three passes required. However, anything beyond this causes the algorithm to grow exponentially in time complexity.

Implementation examples for Bubble sort can be found in A.1.

1.3 Bubble Sort

Best-Case Performance	$O(n)$
Average Performance	$O(n^2)$
Worst-Case Performance	$O(n^2)$
Worst-Case Space Complexity	$O(1)$

1.3.1 Introduction

Bubble sort is an algorithm that goes item by item, sorting pairs of two adjacent items at a time. This is where the name *Bubble sort* originates from, due to the fact that the algorithm creates bubbles around two adjacent items at a time and only considers the order of these two. This is one of simplest comparison sorting algorithms, as it can be implemented in a few number of lines of code, and is simple to conceptualize. The algorithm runs through the list of items swapping pairs as needed, until no more swaps were made on a single pass, which would mean that each item is in its position.

The worst case complexity of bubble sort is $O(n^2)$, because each item can only be moved one position in a pass, meaning that for each item that is out of position must be slowly moved into position. For even relatively small lists of data, this can take times which are far too large to be useful beyond a demonstration. There is one case where bubble sort can provide an advantage over other sorting algorithms. Because the entire process of the sorting, is checking if the items are in a sorted order, if the list is already in a sorted order, Bubble sort will make a single pass testing if everything is in a sorted order. After it finishes the single pass it will determine that the list is already sorted, and end.

Bubble sort has two elements that cause sorting to be extremely slowed. These are “turtles”, and “rabbits”.

“Rabbits” are elements that are at the beginning of the list and belong at the end of the list. Because bubble sort moves from the first to the last element on a repeating order, these “rabbits” will quickly move from the front to the

end, because everytime they are compared to a another value, they are found to be greater than that value, and are swapped, until they reach their final position. They are called “rabbits” due to the fact that they will move to their final destination on the first pass, unless a “faster rabbit” is found. Below is an example of a “rabbit”, where 9 is the “rabbit”.

```

925580
925580
295580
259580
255980
255890
255809

```

“Turtles” are items that are near the end of the list, that belong at the beginning. These are called “turtles” due to the fact that they cause the algorithm to run for much longer, and cannot move quickly to their final destination. The “turtle” can only move once per pass, as they are only moved for a single comparison, before the bubble moves on to the next adjacent pair. These “turtles” are what commonly cause the worst case performance that is found in Bubble sort. As they can strictly only move one position for each pass, they can at most require $n - 1$ passes to move to their final position, where n is the number of terms in the list. Below is an example of a “turtle”, where 0 is the “turtle”. Each row is a new pass of the algorithm, where the bubble has passed through all the elements, similar to that demonstrated in the example for

“rabbits”, essentially there are 5 steps between each step shown in the following example.

```

          925580
1st...255890
2ed...255809
3ed...255089
4th...250589
5th...205589
          025589

```

1.3.2 Algorithm

Algorithm 1.3.C Bubble Sort

```

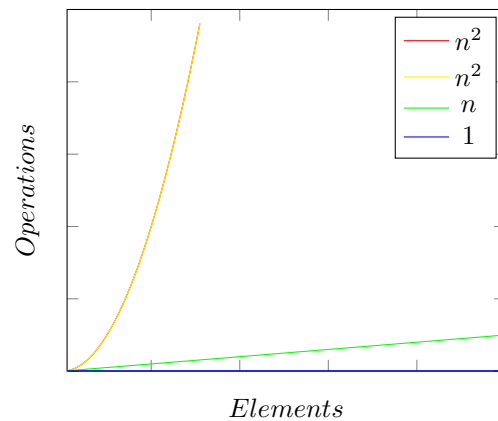
function BUBBLE(a)
  length  $\leftarrow$  length(a)
  while length  $\neq$  0 do
    newLength  $\leftarrow$  0
    for i = 1 to length - 1 do
      if a[i - 1] > a[i] then
        Swap(a[i - 1], a[i])
        newLength  $\leftarrow$  i
      end if
    end for
    length  $\leftarrow$  newLength
  end while
  return a
end function

```

This is a simple pseudocode algorithm for a bubble sort implementation. The algorithm begins by getting a value for the length of the array (*length*). Then the algorithm runs while this value is not equal to zero. Every time the algorithm begins a new pass through the array, it creates a new length value (*newLength*). Then the algorithm runs through the values of the array from the first value to the value at *length*. The *newLength* value is set to the position of the last swap. This is due to the fact that beyond that position, there where no new swaps, so

the data must already be sorted. This practice is to remove some additional time, as there is no reason for the algorithm to continue to check the already sorted data. The *length* value is set to the value of *newLength*, then the process begins. As the value of *length* becomes shorter, the algorithm is getting closer to completion. When the value of *length* becomes 0, this means that there were no more swaps in the list, and that must imply that all the values have been sorted correctly.

1.3.3 Analysis



This is a plot for the theoretical run times that Bubble sort should provide based on the number of elements provided. As it can be seen the best and average cases (which are the same) grow extremely rapidly. The the rare best case preformance (which only will occur if a single pass is necessary, e.g. the array is already sorted) is much lower. The memory usage is constant at 1 as the sorting algorithm does not need to store or make copies of the data, simply to view and make swaps to the data. The exponential nature of the average case time complexity, means that this algorithm will rapidly grow to an unsuable length of time required to run. As it can clearly be seen in the implementation data below.

1.3.4 Conclusion

The bubble sort algorithm, is extremely inefficient, and should only be used for demonstrational purposes, or if the data is near the correct order. Because the algorithm is effective at checking the set of data, and can simply move a

position one or two spots with only a maximum of two or three passes required. However, anything beyond this causes the algorithm to grow exponentially in time complexity.

Implementation examples for Bubble sort can be found in A.1.

1.4 Bubble Sort

Best-Case Performance	$O(n)$
Average Performance	$O(n^2)$
Worst-Case Performance	$O(n^2)$
Worst-Case Space Complexity	$O(1)$

1.4.1 Introduction

Bubble sort is an algorithm that goes item by item, sorting pairs of two adjacent items at a time. This is where the name *Bubble sort* originates from, due to the fact that the algorithm creates bubbles around two adjacent items at a time and only considers the order of these two. This is one of simplest comparison sorting algorithms, as it can be implemented in a few number of lines of code, and is simple to conceptualize. The algorithm runs through the list of items swapping pairs as needed, until no more swaps were made on a single pass, which would mean that each item is in its position.

The worst case complexity of bubble sort is $O(n^2)$, because each item can only be moved one position in a pass, meaning that for each item that is out of position must be slowly moved into position. For even relatively small lists of data, this can take times which are far too large to be useful beyond a demonstration. There is one case where bubble sort can provide an advantage over other sorting algorithms. Because the entire process of the sorting, is checking if the items are in a sorted order, if the list is already in a sorted order, Bubble sort will make a single pass testing if everything is in a sorted order. After it finishes the single pass it will determine that the list is already sorted, and end.

Bubble sort has two elements that cause sorting to be extremely slowed. These are “turtles”, and “rabbits”.

“Rabbits” are elements that are at the beginning of the list and belong at the end of the list. Because bubble sort moves from the first to the last element on a repeating order, these “rabbits” will quickly move from the front to the

end, because everytime they are compared to a another value, they are found to be greater than that value, and are swapped, until they reach their final position. They are called “rabbits” due to the fact that they will move to their final destination on the first pass, unless a “faster rabbit” is found. Below is an example of a “rabbit”, where 9 is the “rabbit”.

```

925580
925580
295580
259580
255980
255890
255809

```

“Turtles” are items that are near the end of the list, that belong at the beginning. These are called “turtles” due to the fact that they cause the algorithm to run for much longer, and cannot move quickly to their final destination. The “turtle” can only move once per pass, as they are only moved for a single comparison, before the bubble moves on to the next adjacent pair. These “turtles” are what commonly cause the worst case performance that is found in Bubble sort. As they can strictly only move one position for each pass, they can at most require $n - 1$ passes to move to their final position, where n is the number of terms in the list. Below is an example of a “turtle”, where 0 is the “turtle”. Each row is a new pass of the algorithm, where the bubble has passed through all the elements, similar to that demonstrated in the example for

“rabbits”, essentially there are 5 steps between each step shown in the following example.

```

          925580
1st...255890
2ed...255809
3ed...255089
4th...250589
5th...205589
          025589

```

1.4.2 Algorithm

Algorithm 1.4.D Bubble Sort

```

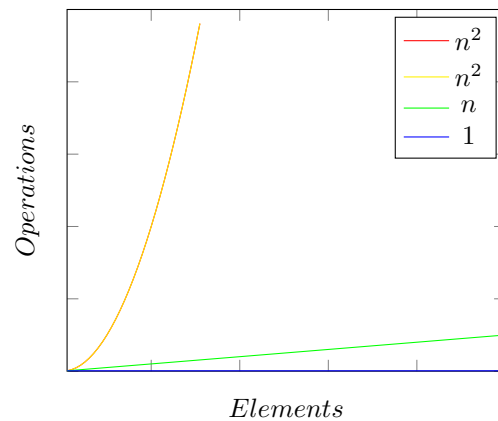
function BUBBLE(a)
  length  $\leftarrow$  length(a)
  while length  $\neq$  0 do
    newLength  $\leftarrow$  0
    for i = 1 to length - 1 do
      if a[i - 1] > a[i] then
        Swap(a[i - 1], a[i])
        newLength  $\leftarrow$  i
      end if
    end for
    length  $\leftarrow$  newLength
  end while
  return a
end function

```

This is a simple pseudocode algorithm for a bubble sort implementation. The algorithm begins by getting a value for the length of the array (*length*). Then the algorithm runs while this value is not equal to zero. Every time the algorithm begins a new pass through the array, it creates a new length value (*newLength*). Then the algorithm runs through the values of the array from the first value to the value at *length*. The *newLength* value is set to the position of the last swap. This is due to the fact that beyond that position, there where no new swaps, so

the data must already be sorted. This practice is to remove some additional time, as there is no reason for the algorithm to continue to check the already sorted data. The *length* value is set to the value of *newLength*, then the process begins. As the value of *length* becomes shorter, the algorithm is getting closer to completion. When the value of *length* becomes 0, this means that there were no more swaps in the list, and that must imply that all the values have been sorted correctly.

1.4.3 Analysis



This is a plot for the theoretical run times that Bubble sort should provide based on the number of elements provided. As it can be seen the best and average cases (which are the same) grow extremely rapidly. The the rare best case preformance (which only will occur if a single pass is necessary, e.g. the array is already sorted) is much lower. The memory usage is constant at 1 as the sorting algorithm does not need to store or make copies of the data, simply to view and make swaps to the data. The exponential nature of the average case time complexity, means that this algorithm will rapidly grow to an unsuable length of time required to run. As it can clearly be seen in the implementation data below.

1.4.4 Conclusion

The bubble sort algorithm, is extremely inefficient, and should only be used for demonstrational purposes, or if the data is near the correct order. Because the algorithm is effective at checking the set of data, and can simply move a

position one or two spots with only a maximum of two or three passes required. However, anything beyond this causes the algorithm to grow exponentially in time complexity.

Implementation examples for Bubble sort can be found in A.1.

1.5 Bubble Sort

Best-Case Performance	$O(n)$
Average Performance	$O(n^2)$
Worst-Case Performance	$O(n^2)$
Worst-Case Space Complexity	$O(1)$

1.5.1 Introduction

Bubble sort is an algorithm that goes item by item, sorting pairs of two adjacent items at a time. This is where the name *Bubble sort* originates from, due to the fact that the algorithm creates bubbles around two adjacent items at a time and only considers the order of these two. This is one of simplest comparison sorting algorithms, as it can be implemented in a few number of lines of code, and is simple to conceptualize. The algorithm runs through the list of items swapping pairs as needed, until no more swaps were made on a single pass, which would mean that each item is in its position.

The worst case complexity of bubble sort is $O(n^2)$, because each item can only be moved one position in a pass, meaning that for each item that is out of position must be slowly moved into position. For even relatively small lists of data, this can take times which are far too large to be useful beyond a demonstration. There is one case where bubble sort can provide an advantage over other sorting algorithms. Because the entire process of the sorting, is checking if the items are in a sorted order, if the list is already in a sorted order, Bubble sort will make a single pass testing if everything is in a sorted order. After it finishes the single pass it will determine that the list is already sorted, and end.

Bubble sort has two elements that cause sorting to be extremely slowed. These are “turtles”, and “rabbits”.

“Rabbits” are elements that are at the beginning of the list and belong at the end of the list. Because bubble sort moves from the first to the last element on a repeating order, these “rabbits” will quickly move from the front to the

end, because everytime they are compared to a another value, they are found to be greater than that value, and are swapped, until they reach their final position. They are called “rabbits” due to the fact that they will move to their final destination on the first pass, unless a “faster rabbit” is found. Below is an example of a “rabbit”, where 9 is the “rabbit”.

```

925580
925580
295580
259580
255980
255890
255809

```

“Turtles” are items that are near the end of the list, that belong at the beginning. These are called “turtles” due to the fact that they cause the algorithm to run for much longer, and cannot move quickly to their final destination. The “turtle” can only move once per pass, as they are only moved for a single comparison, before the bubble moves on to the next adjacent pair. These “turtles” are what commonly cause the worst case performance that is found in Bubble sort. As they can strictly only move one position for each pass, they can at most require $n - 1$ passes to move to their final position, where n is the number of terms in the list. Below is an example of a “turtle”, where 0 is the “turtle”. Each row is a new pass of the algorithm, where the bubble has passed through all the elements, similar to that demonstrated in the example for

“rabbits”, essentially there are 5 steps between each step shown in the following example.

```

          925580
1st...255890
2ed...255809
3ed...255089
4th...250589
5th...205589
          025589

```

1.5.2 Algorithm

Algorithm 1.5.E Bubble Sort

```

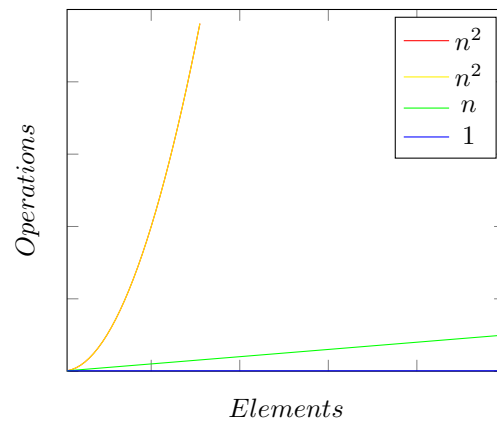
function BUBBLE(a)
  length  $\leftarrow$  length(a)
  while length  $\neq$  0 do
    newLength  $\leftarrow$  0
    for i = 1 to length - 1 do
      if a[i - 1] > a[i] then
        Swap(a[i - 1], a[i])
        newLength  $\leftarrow$  i
      end if
    end for
    length  $\leftarrow$  newLength
  end while
  return a
end function

```

This is a simple pseudocode algorithm for a bubble sort implementation. The algorithm begins by getting a value for the length of the array (*length*). Then the algorithm runs while this value is not equal to zero. Every time the algorithm begins a new pass through the array, it creates a new length value (*newLength*). Then the algorithm runs through the values of the array from the first value to the value at *length*. The *newLength* value is set to the position of the last swap. This is due to the fact that beyond that position, there where no new swaps, so

the data must already be sorted. This practice is to remove some additional time, as there is no reason for the algorithm to continue to check the already sorted data. The *length* value is set to the value of *newLength*, then the process begins. As the value of *length* becomes shorter, the algorithm is getting closer to completion. When the value of *length* becomes 0, this means that there were no more swaps in the list, and that must imply that all the values have been sorted correctly.

1.5.3 Analysis



This is a plot for the theoretical run times that Bubble sort should provide based on the number of elements provided. As it can be seen the best and average cases (which are the same) grow extremely rapidly. The the rare best case preformance (which only will occure if a single pass is neccasary, e.g. the array is already sorted) is much lower. The memory usage is constant at 1 as the sorting algorithm does not need to store or make copies of the data, simply to view and make swaps to the data. The exponential nature of the average case time complexity, means that this algorithm will rapidly grow to an unsuable length of time required to run. As it can clearly be seen in the implementation data below.

1.5.4 Conclusion

The bubble sort algorithm, is extremely inefficient, and should only be used for demonstrational purposes, or if the data is near the correct order. Because the algorithm is effective at checking the set of data, and can simply move a

position one or two spots with only a maximum of two or three passes required. However, anything beyond this causes the algorithm to grow exponentially in time complexity.

Implementation examples for Bubble sort can be found in A.1.

1.6 Bubble Sort

Best-Case Performance	$O(n)$
Average Performance	$O(n^2)$
Worst-Case Performance	$O(n^2)$
Worst-Case Space Complexity	$O(1)$

1.6.1 Introduction

Bubble sort is an algorithm that goes item by item, sorting pairs of two adjacent items at a time. This is where the name *Bubble sort* originates from, due to the fact that the algorithm creates bubbles around two adjacent items at a time and only considers the order of these two. This is one of simplest comparison sorting algorithms, as it can be implemented in a few number of lines of code, and is simple to conceptualize. The algorithm runs through the list of items swapping pairs as needed, until no more swaps were made on a single pass, which would mean that each item is in its position.

The worst case complexity of bubble sort is $O(n^2)$, because each item can only be moved one position in a pass, meaning that for each item that is out of position must be slowly moved into position. For even relatively small lists of data, this can take times which are far too large to be useful beyond a demonstration. There is one case where bubble sort can provide an advantage over other sorting algorithms. Because the entire process of the sorting, is checking if the items are in a sorted order, if the list is already in a sorted order, Bubble sort will make a single pass testing if everything is in a sorted order. After it finishes the single pass it will determine that the list is already sorted, and end.

Bubble sort has two elements that cause sorting to be extremely slowed. These are “turtles”, and “rabbits”.

“Rabbits” are elements that are at the beginning of the list and belong at the end of the list. Because bubble sort moves from the first to the last element on a repeating order, these “rabbits” will quickly move from the front to the

end, because everytime they are compared to a another value, they are found to be greater than that value, and are swapped, until they reach their final position. They are called “rabbits” due to the fact that they will move to their final destination on the first pass, unless a “faster rabbit” is found. Below is an example of a “rabbit”, where 9 is the “rabbit”.

```

925580
925580
295580
259580
255980
255890
255809

```

“Turtles” are items that are near the end of the list, that belong at the beginning. These are called “turtles” due to the fact that they cause the algorithm to run for much longer, and cannot move quickly to their final destination. The “turtle” can only move once per pass, as they are only moved for a single comparison, before the bubble moves on to the next adjacent pair. These “turtles” are what commonly cause the worst case performance that is found in Bubble sort. As they can strictly only move one position for each pass, they can at most require $n - 1$ passes to move to their final position, where n is the number of terms in the list. Below is an example of a “turtle”, where 0 is the “turtle”. Each row is a new pass of the algorithm, where the bubble has passed through all the elements, similar to that demonstrated in the example for

“rabbits”, essentially there are 5 steps between each step shown in the following example.

```

          925580
1st...255890
2ed...255809
3ed...255089
4th...250589
5th...205589
          025589

```

1.6.2 Algorithm

Algorithm 1.6.F Bubble Sort

```

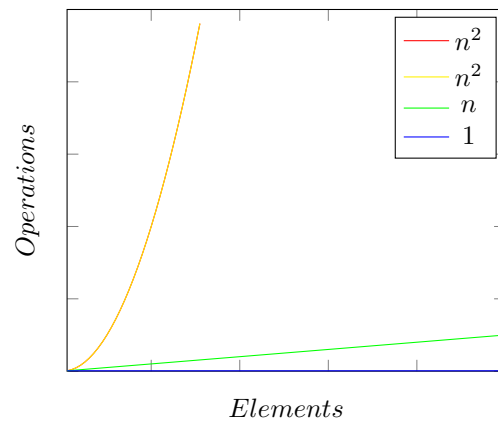
function BUBBLE(a)
  length  $\leftarrow$  length(a)
  while length  $\neq$  0 do
    newLength  $\leftarrow$  0
    for i = 1 to length - 1 do
      if a[i - 1] > a[i] then
        Swap(a[i - 1], a[i])
        newLength  $\leftarrow$  i
      end if
    end for
    length  $\leftarrow$  newLength
  end while
  return a
end function

```

This is a simple pseudocode algorithm for a bubble sort implementation. The algorithm begins by getting a value for the length of the array (*length*). Then the algorithm runs while this value is not equal to zero. Every time the algorithm begins a new pass through the array, it creates a new length value (*newLength*). Then the algorithm runs through the values of the array from the first value to the value at *length*. The *newLength* value is set to the position of the last swap. This is due to the fact that beyond that position, there where no new swaps, so

the data must already be sorted. This practice is to remove some additional time, as there is no reason for the algorithm to continue to check the already sorted data. The *length* value is set to the value of *newLength*, then the process begins. As the value of *length* becomes shorter, the algorithm is getting closer to completion. When the value of *length* becomes 0, this means that there were no more swaps in the list, and that must imply that all the values have been sorted correctly.

1.6.3 Analysis



This is a plot for the theoretical run times that Bubble sort should provide based on the number of elements provided. As it can be seen the best and average cases (which are the same) grow extremely rapidly. The the rare best case preformance (which only will occur if a single pass is necessary, e.g. the array is already sorted) is much lower. The memory usage is constant at 1 as the sorting algorithm does not need to store or make copies of the data, simply to view and make swaps to the data. The exponential nature of the average case time complexity, means that this algorithm will rapidly grow to an unsuable length of time required to run. As it can clearly be seen in the implementation data below.

1.6.4 Conclusion

The bubble sort algorithm, is extremely inefficient, and should only be used for demonstrational purposes, or if the data is near the correct order. Because the algorithm is effective at checking the set of data, and can simply move a

position one or two spots with only a maximum of two or three passes required. However, anything beyond this causes the algorithm to grow exponentially in time complexity.

Implementation examples for Bubble sort can be found in A.1.

Appendices

A Algorithms

A.1 Bubble Sort

A.1.1 C++

```
1 #include <array>
2 #include <vector>
3
4 namespace bubble {
5     std::array<int, 2> Run();
6 } // namespace bubble
```

appendix/bubble.hpp

```
1 #include "algo/bubble.hpp"
2
3 #include <array>
4 #include <vector>
5
6 #include "sort.hpp"
7
8 std::array<int, 2> bubble::Run() {
9     int length = data.size();
10    std::array<int, 2> track = {{0, 0}};
11    while (length != 0) {
12        int new_length = 0;
13        for (int i = 1; i < length; i++) {
14            track[0] += 2;
15            if (data[i - 1] > data[i]) {
16                track[1]++;
17                iter_swap(data.begin() + i - 1, data.begin() + i);
18                new_length = i;
19            }
20        }
21        length = new_length;
22    }
23    return track;
24 }
```

appendix/bubble.cpp