Sorting Algorithms

Arden Rasmussen

September 29, 2017

Contents

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 beable to utilize the algoritalgorithm that is best suited for a data set, instead of applying a general solution that will at a sub optimal rate. This paper aims to calsify a wide selecton of sorting algorithms for the different data sets that they are most optimaly used in conjunction with.

1 Comparison Sorts

Comparison sorting algorithms, are algortims that directly compare the entrys 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 swaped to further the order of the array of items, untill the array is compleatly sorted. These sorting algorithms are some of the msot common, becbecause it is fairly simple to implement, and to theoreticaly think of.

This means that there is a large number of compairson sorts, that are commonlcommonly used in the world of computer science. Some of the most commocommon are Bubble sort, Quicksort or Merge sort. One aspect of compaisoncompaison sort algorithms is that they tend to use less memory, thanthan that of non comparison sorting algorithms, because the comparison sotrs tend to swap the elements of the list in place, insetead of creating new lists, and copying. However, they tend to be limited because they are dirdirrectly comparing each elemnt to one another.

Contents

1.1 Binary Tree Sort

Best-Case Preformance	$O(n \log n)$
Average Preformance	$O(n \log n)$
Worst-Case Preformance	$O(n^2)$
Worst-Case Space Complexity	$O(n)$

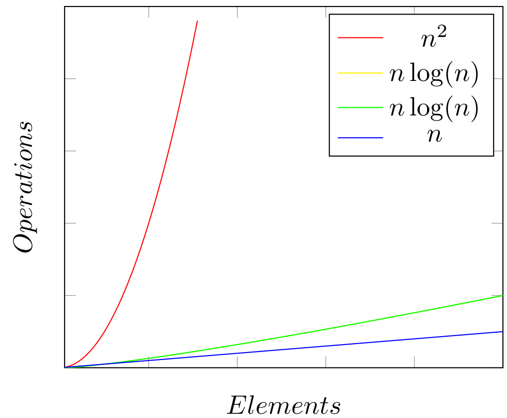
1.1.1 Introduction

1.1.2 Algorithm

Algorithm 1.1.A Bubble Sort

function BUBBLE(a)
  $length$ ←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

1.1.3 Analysis



1.1.4 Conclusion

Implementation examples for Binary Tree sort can be found in **??**.

1.2 Bubble Sort

Best-Case Preformance	$O(n)$
Average Preformance	$O(n^2)$
Worst-Case Preformance	$O(n^2)$
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 conly considers the order of these two. This is one of simplest comprison sorting algorithms, as it can be inplemented in a few number o flines of code, and is simple to conceptualize. The algorithm runs through the list of items swaping pairs as needed, untill 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 positon in a pass, meaning that for each item that is out of position must be slowly moved into position. For even relativly small lists of data, this can take times which are far to 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 sotring, 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 determin that the list is already sorted, and end.

Bubble sort has two elements that cause sorting to be extreamly slowed. These are "turtles", and "rabbits".

"Rabbits" are elements that are at the beginnging 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 compaired to a nother value, they are found to be greater than that value, and are swaped, until they reach their final position. They are called "rabbits" due to the fact that they will move to their final destination on the furst 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 begining. 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 preformance that is found in Bubble sort. As they can strictly only move one positon 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 elemnts, similar to that demonstrated in the example for

"rabbits", essentialy there are 5 steps between each step shown in the folowing example.

$$\begin{aligned} & 925580 \\ 1st \cdots & 255890 \\ 2ed \cdots & 255809 \\ 3ed \cdots & 255089 \\ 4th \cdots & 250589 \\ 5th \cdots & 205589 \\ & 025589 \end{aligned}$$

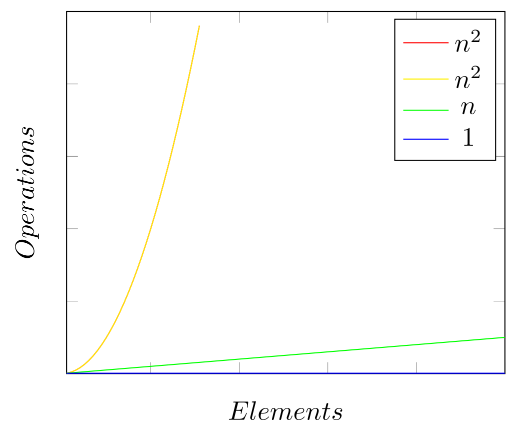
1.2.2 Algorithm

Algorithm 1.2.B Bubble Sort

```
function BUBBLE(a)
    length ←length(a)
    while length ≠ 0 do
        newLength ← 0
        for i = 1 to length − 1 do
            if a[i − 1] > a[i] then
                Swap(a[i − 1],a[i])
                newLength ← i
            end if
        end for
        length ← 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, is 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 compleation. 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 correclty.

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 extreamly rappidly. 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 rappidly 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 extreamly inefficient, and should only be used for demonstrational proupuses, 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 exponentialy in time complexity.

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

1.3 Bubble Sort

Best-Case Preformance	$O(n)$
Average Preformance	$O(n^2)$
Worst-Case Preformance	$O(n^2)$
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 conly considers the order of these two. This is one of simplest comprison sorting algorithms, as it can be inplemented in a few number o flines of code, and is simple to conceptualize. The algorithm runs through the list of items swaping pairs as needed, untill 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 positon in a pass, meaning that for each item that is out of position must be slowly moved into position. For even relativly small lists of data, this can take times which are far to 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 sotring, 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 determin that the list is already sorted, and end.

Bubble sort has two elements that cause sorting to be extreamly slowed. These are "turtles", and "rabbits".

"Rabbits" are elements that are at the beginnging 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 compaired to a nother value, they are found to be greater than that value, and are swaped, until they reach their final position. They are called "rabbits" due to the fact that they will move to their final destination on the furst 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 begining. 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 preformance that is found in Bubble sort. As they can strictly only move one positon 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 elemnts, similar to that demonstrated in the example for

"rabbits", essentialy there are 5 steps between each step shown in the folowing example.

$$925580$$
$$1st \cdots 2558\textcolor{red}{90}$$
$$2ed \cdots 255\textcolor{red}{80}9$$
$$3ed \cdots 25\textcolor{red}{50}89$$
$$4th \cdots 2\textcolor{red}{50}589$$
$$5th \cdots \textcolor{red}{20}5589$$
$$025589$$

1.3.2 Algorithm

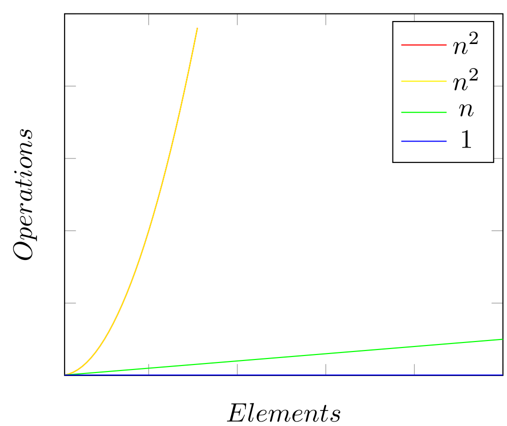
Algorithm 1.3.C Bubble Sort

```
function BUBBLE(a)
    length ←length (a)
    while length ≠ 0 do
        newLength ← 0
        for i = 1 to length − 1 do
            if a[i − 1] > a[i] then
                Swap (a[i − 1],a[i])
                newLength ← i
            end if
        end for
        length ← newLength
    end while
    return a
end function
```

$$\frac{7}{8} * 16 thisismath \tag{1}$$

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, is 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 compleation. 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 correclty.

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 extreamly rappidly. 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 rappidly 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 extreamly inefficient, and should only be used for demonstrational proupuses, 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 exponentialy in time complexity.

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

1.4 Bubble Sort

Best-Case Preformance	$O(n)$
Average Preformance	$O(n^2)$
Worst-Case Preformance	$O(n^2)$
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 conly considers the order of these two. This is one of simplest comprison sorting algorithms, as it can be inplemented in a few number o flines of code, and is simple to conceptualize. The algorithm runs through the list of items swaping pairs as needed, untill 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 positon in a pass, meaning that for each item that is out of position must be slowly moved into position. For even relativly small lists of data, this can take times which are far to 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 sotring, 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 determin that the list is already sorted, and end.

Bubble sort has two elements that cause sorting to be extreamly slowed. These are "turtles", and "rabbits".

"Rabbits" are elements that are at the beginnging 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 compaired to a nother value, they are found to be greater than that value, and are swaped, until they reach their final position. They are called "rabbits" due to the fact that they will move to their final destination on the furst 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 begining. 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 preformance that is found in Bubble sort. As they can strictly only move one positon 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 elemnts, similar to that demonstrated in the example for

"rabbits", essentialy there are 5 steps between each step shown in the folowing example.

$$\begin{aligned} &925580 \\ 1st \cdots\ &255890 \\ 2ed \cdots\ &255809 \\ 3ed \cdots\ &255089 \\ 4th \cdots\ &250589 \\ 5th \cdots\ &205589 \\ &025589 \end{aligned}$$

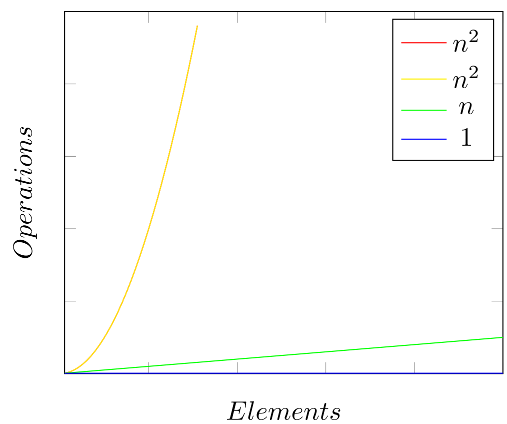
1.4.2 Algorithm

Algorithm 1.4.D Bubble Sort

```
function BUBBLE(a)
    length ←length(a)
    while length ≠ 0 do
        newLength ← 0
        for i = 1 to length − 1 do
            if a[i − 1] > a[i] then
                Swap(a[i − 1],a[i])
                newLength ← i
            end if
        end for
        length ← 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, is 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 compleation. 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 correclty.

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 extreamly rappidly. 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 rappidly 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 extreamly inefficient, and should only be used for demonstrational proupuses, 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 exponentialy in time complexity.

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

1.5 Bubble Sort

Best-Case Preformance	$O(n)$
Average Preformance	$O(n^2)$
Worst-Case Preformance	$O(n^2)$
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 conly considers the order of these two. This is one of simplest comprison sorting algorithms, as it can be inplemented in a few number o flines of code, and is simple to conceptualize. The algorithm runs through the list of items swaping pairs as needed, untill 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 positon in a pass, meaning that for each item that is out of position must be slowly moved into position. For even relativly small lists of data, this can take times which are far to 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 sotring, 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 determin that the list is already sorted, and end.

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

“Rabbits” are elements that are at the beginnging 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 compaired to a nother value, they are found to be greater than that value, and are swaped, until they reach their final position. They are called “rabbits” due to the fact that they will move to their final destination on the furst 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 begining. 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 preformance that is found in Bubble sort. As they can strictly only move one positon 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 elemnts, similar to that demonstrated in the example for

"rabbits", essentialy there are 5 steps between each step shown in the folowing example.

$$\begin{aligned} & 925580 \\ 1st \cdots & 255890 \\ 2ed \cdots & 255809 \\ 3ed \cdots & 255089 \\ 4th \cdots & 250589 \\ 5th \cdots & 205589 \\ & 025589 \end{aligned}$$

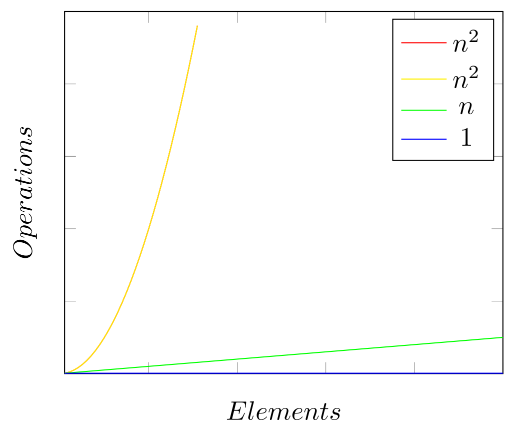
1.5.2 Algorithm

Algorithm 1.5.E Bubble Sort

```
function BUBBLE(a)
    length ←length(a)
    while length ≠ 0 do
        newLength ← 0
        for i = 1 to length − 1 do
            if a[i − 1] > a[i] then
                Swap(a[i − 1],a[i])
                newLength ← i
            end if
        end for
        length ← 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, is 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 compleation. 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 correclty.

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 extreamly rappidly. 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 rappidly 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 extreamly inefficient, and should only be used for demonstrational proupuses, 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 exponentialy in time complexity.

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

1.6 Bubble Sort

Best-Case Preformance	$O(n)$
Average Preformance	$O(n^2)$
Worst-Case Preformance	$O(n^2)$
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 conly considers the order of these two. This is one of simplest comprison sorting algorithms, as it can be inplemented in a few number o flines of code, and is simple to conceptualize. The algorithm runs through the list of items swaping pairs as needed, untill 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 positon in a pass, meaning that for each item that is out of position must be slowly moved into position. For even relativly small lists of data, this can take times which are far to 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 sotring, 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 determin that the list is already sorted, and end.

Bubble sort has two elements that cause sorting to be extreamly slowed. These are "turtles", and "rabbits".

"Rabbits" are elements that are at the beginnging 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 compaired to a nother value, they are found to be greater than that value, and are swaped, until they reach their final position. They are called "rabbits" due to the fact that they will move to their final destination on the furst 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 begining. 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 preformance that is found in Bubble sort. As they can strictly only move one positon 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 elemnts, similar to that demonstrated in the example for

"rabbits", essentialy there are 5 steps between each step shown in the folowing example.

925580
$1st \cdots$ 255890
$2ed \cdots$ 255809
$3ed \cdots$ 255089
$4th \cdots$ 250589
$5th \cdots$ 205589
025589

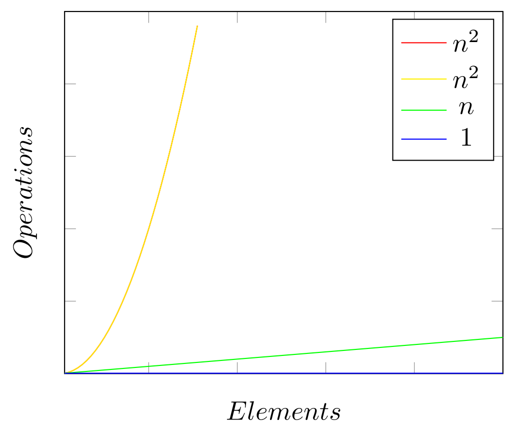
1.6.2 Algorithm

Algorithm 1.6.F Bubble Sort

```
function BUBBLE(a)
    length ←length(a)
    while length ≠ 0 do
        newLength ← 0
        for i = 1 to length − 1 do
            if a[i − 1] > a[i] then
                Swap(a[i − 1],a[i])
                newLength ← i
            end if
        end for
        length ← 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, is 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 compleation. 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 correclty.

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 extreamly rappidly. 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 rappidly 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 extreamly inefficient, and should only be used for demonstrational proupuses, 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 exponentialy 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++

```cpp
#include <array>
#include <vector>

namespace bubble {
  std::array<int, 2> Run();
}  // namespace bubble
```

appendix/bubble.hpp

```cpp
#include "algo/bubble.hpp"

#include <array>
#include <vector>

#include "sort.hpp"

std::array<int, 2> bubble::Run() {
  int length = data.size();
  std::array<int, 2> track = {{0, 0}};
  while (length != 0) {
    int new_length = 0;
    for (int i = 1; i < length; i++) {
      track[0] += 2;
      if (data[i - 1] > data[i]) {
        track[1]++;
        iter_swap(data.begin() + i - 1, data.begin() + i);
        new_length = i;
      }
    }
    length = new_length;
  }
  return track;
}
```

appendix/bubble.cpp