Different types of sorting algorithms

Josh Crail
Undergrad CSE student
Miami University
crailjc@miamioh.edu

May 2021

1 Introduction

In the context of computer science sorting is one of the most important concepts for any data to usable or readable in any way. Without sorting any data set would take a sequential search to even find what is needed. With sorting data can be taken and then used efficiently by some program to carry out some task. Without sorting the ability to do any task would be hindered by a sequential search of a data set looking for a value; this is inherently against the idea that computer science is all about efficiency. Without sorting doing something as simple as searching Twitter for a single tweet could take hours as it has to go through every tweet as there has been no sorting applied. Even more important that just sorting is efficient sorting as everyone always wants things to be fast, no one want to wait any longer than what they have to. As such if a sorting algorithm is not efficient or its something that's not right it will not be fast enough, which is important because regardless how fast a computer is a bad algorithm will only slow things down, and in the world of computer science speed is always important and any minor increase in efficiency could drastically reduce how long an algorithm takes on big data. The purpose of this report is to introduce six uncommon, but still important, sorting algorithms. Along with this each sorting algorithm will have an in depth explanation of itself covering: its concept, its class, pseudo code, example of its operation, time complexity, notable special properties, practical application, and its implementation in Java. The Pancake sort that is a sorting problem which is used to get different sized stacks in the correct order. The unique part of this algorithm is the fact that the sorting is similar to the Tower of Hanoi problem where the individual pancakes cannot be moved into a place the it requires the stack to be flipped. The Gnome Sort or stupid sort is a sorting algorithm that works with a single item at a time swapping it to the sorted position. What makes this sorting algorithm unique is its ability to be used without nested for loop. This sort stems from a very basic fairy tail that described how a gnome would sort his pots that he had, but his sorting method was a very stupid or inefficient sorting method with little practicality. Tournament sort is a sorting algorithm that works by creating/building an ordered binary tree. With this algorithm it can be used to efficiently find the smallest or largest elements within an array of elements. This overall sorting algorithm comes from the very standard idea of a single elimination tournament where two people face off and the winner will go on and keep facing people until only one is left and a winner is chosen. The Stooge sort is a very inefficient sorting algorithm, that has very little practicality but shows how not all sorting algorithms need to be overly complex to do sorting. Also this sorting algorithm is fairly notable for its homage to the famous slap stick comedy group the three stooges. Patience Sort is a sorting algorithm which the name stems from the card game patience or its more common name solitaire, where cards (elements are dealt into piles). Also what makes this algorithm of note is its ability to compute the longest sub sequence length. The Pigeon sort is a sorting algorithm that comes from a preexisting concept from the real world. Its a fairly unique algorithm that sorts smaller subsections and then then combine them all of them together. Overall this algorithm has a fairly efficient cost for it.

2 Algorithm Details

2.1 Pancake Sort

2.1.1 The Concept

The overall concept behind this is basic but is more challenging when examining it further. The idea of the pancake sort it to take n number of pancakes and sort them such that the pancakes are in order from smallest on top and largest on the bottom. It is believed that this algorithm was though of based off how bread was cooked, specifically how Indian bread was cooked, where its done by putting the bread in a single column and flipping each one so that all of the bread ends up being cooked. Also this problem was originally an open problem for the past thirty years, yet in 2011 it was determined that to find the smallest amount of flips for a stack of pancakes is NP-hard.

2.1.2 Pseudo code

```
1. procedure pankcake_sort(array[]):
2. height = array.length
```

```
3. while (height > 1):
4.   top = getMax(array, height)
5.   if (top != height):
6.     flip(array, top)
7.     flip(array, height-1)
8.   height--
```

```
1. procedure getMax(array[], height):
2. max = 0
3. for(i : array[]):
4.  if (array[i] > array[max]):
5.    max = i
6. return max
```

```
1. procedure flip(array[], n):
2. start = 0
3. while (start < n):
4.   swap(array[start], array[n])
5.   start++
6.   n--
```

2.1.3 Cost Analysis

The overall time for this algorithm ends up being O(n^2) for the average and worst case. The best case will be the stack of pancakes is already sorted as such we will only need to iterate through the pancake array O(n). For the other cases they will end up being O(n^2) because we will have to iterate through the n sized pancake array O(n). While iterating through we will have to do n - 1 flips. Overall the sorting will end up with a time complexity of O(n^2).

2.1.4 Practical Application

Overall this algorithm has many applications where a system has to sort a stack where a system needs to have a stack of something sorted by their size and where a specific side needs to be facing down. For example a company that ships books of various sizes and when the books are being put in the box they are not in order and some are title up while others are not, something like the burnt pancake sort could be useful for something like a robotic arm to sort books this way. Note this is an example that is just to demonstrate a plausible, but theoretical example of the application of the burnt pancake sort.

2.2 Stupid Sort (Gnome Sort)

2.2.1 The Concept

The Gnome Sort or Stupid Sort was first thought of by Hamid Sarbazi-Azad in 2000. The idea behind this Gnome Sort is to find the first location where two neighboring elements are in the incorrect order and Gnome Sort will swap the two elements. With Gnome Sort it will account for the possibility of having a swap that could cause a new out of order neighbor pair next to the newly swapped elements. Gnome Sort will not have any assumptions about elements in front of it being sorted, so it only need to check the position of right before the swapped elements and then keep swapping down until that elements reaches its appropriate spot. The interesting thing to note about this algorithm is that fact that it has no dependencies on nested loops even though its it iterating through a list of items to be sorted. With this unique no nested loop dependencies causes the overall algorithm to be fairly simple or even stupid due to its fairly inefficient nature.

2.2.2 Pseudo code

```
1. procedure gnomeSort(array[]):
2.  i = 0
3.  while (i < length(array)):
4.    if (i = 0 or array[i] >= array[i - 1]):
5.      continue to next element (i++)
6.    else
7.      swap the element array[i]
8.      with the array[i-1] element
9.      then check the previous
10.     element (i--)
```

2.2.3 Cost Analysis

Starting with the best case performance of this algorithm would be that everything is in order and as such Gnome Sort would just iterate through the elements thus being O(n). Both the worst case and average case end up being the same that is O(n^2) this is due to the fact that its unlikely a list will ever be completely sorted as such there will be multiple times where the Gnome sort will have to swap and element multiple times also even if elements are initially sorted the algorithm will still have to go back and check them. As such any potentially sorted items will still need to be checked along with having to iterate through an get an element to its correct position.

2.2.4 Practical Application

Due to the very inefficient nature of Gnome sort it has no real practical applications. With optimization Gnome sort could have a variable to store its position before traversing back, which would only cause it to become an algorithm similar to insertion sort. Overall this algorithm is meant more for demonstration purposes rather than any real world application hence why its also referred to as a stupid sort.

2.3 Tournament Sort

2.3.1 The Concept

Tournament sort (a fast variant of the heap sort) like the name implies comes from the idea of how a single elimination tournament is done, one on one with successive rounds until a 'winner' is found. As such due to the common and easy to understand concept and that tournaments have been happening long be before algorithms were being made. The concept is not able to be really attributed as someone thinking of it, its more

someone taking the concept and making the algorithm for it. The first example of tournament sort is from Robert Floyd in 1962, creating the tournament sort algorithm that is like how a single elimination tournament is played out. Basically how the Tournament sort works is by having successive rounds where two values will face off and either the greater or lesser of those values will go on to the next round and this process continues for each successive round until a winner is found. To note what needs to be remembered is that while a max or min can be found the value in second place is not guaranteed to be the second greatest or least value as the winner value could of already faced value that may of even beat the value that is in second place. As such this is ran multiple time each time the winner is taken out and at the end when there is a single element left that is inserted at the end and then the values have been sorted either by max or by min.

2.3.2 Pseudo code

In this case we are finding the min with this tournament sort

```
1. procedure TournamentSort(array[]):
2.   while (length(array) >1):
3.       temparray = array
4.     while (length(temparray) > 1):
5.       compare pairs of elements
6.       losing elements are removed
7.       FindMin(array[i], array[i+1])
8.       winning elements compared again
9.    add winner to result and
      remove from array

1.  procedure FindMin(x,y):
2.       if (x > y):
3.           return y
4.       else
5.           return x
```

2.3.3 Cost Analysis

Due to the nature of this algorithm it will always have to go through every element to find either the max or min for each iteration as such its performance will remain as a constant O(n log n) as this algorithm operates the same regardless if the elements are already sorted or not. It will take O(n) to select the next element from the array, but as already mentioned it will take O(n log n) operations after the 'tournament' has been built O(n).

2.3.4 Practical Application

Over all its practical application is useful in circumstances where you needs a sorting algorithm with O(n log n) time that is used to find the max or min of some object. One of its more useful applications is it can be used in Multi-way merges (taking m sorted lists and merging those lists into a singular sorted list). Also another use of this algorithm is to get initial runs for external sorting algorithms.

2.4 Pigeonhole Sort

2.4.1 The Concept

The name pigeonhole comes from the idea in mathematics of the same name. The concept in mathematics is there are n items being put into m containers where n will always be greater than m as such at least one container will contain more than one. The real world example of this are pigeonholes where pigeons will roost in a hole even if another pigeon is in that hole. The sorting concept is really the same where there is an array that needs to be sorted by the available slots (pigeonholes, m) are less than the elements in the array (n). Every pigeonhole has a key similar to the value going to be put into the array. For example a pigeonhole key of three would hold elements with a value of three an example would be a string with a length of three. When elements are being inserted they will be inserted in such a way that they will be inserted a way such that every hole will be sorted and as such the whole array will be sorted. One aspect that needs to be remembered is that values should be sufficiently close together meaning that array values that are not close together such as 1, 200, 10000 means that it would be very inefficient as there will be no overlap in values.

2.4.2 Pseudo code

```
1. Procedure pigeonhole_sort (array):
2. min = array[0], max = array[0]
3.  for(i : array):
4.   maxVal = max(array[i], max)
5.   minVal = min(array[i], min)
6.
7.  range = maxVal - minVal + 1
8.  pigeonArr[range]
9.
10. for (i : array):
11.   pigeonArr[array[i] - min]++
12.
13. index = 0
14. for(i : range):
15.     while(pigeonArr[i]-->0):
16.         array[index++]=i+min
```

2.4.3 Cost Analysis

The cost for this sorting algorithm is O(n+N) where N is the range of the keys and n is the size of the array. This will be the worse case performance, this time is because all items are moved twice the first to the pigeon array and second when its moved to its final sorted location at the end

2.4.4 Practical Application

The overall application for this algorithm pretty standard and the overall requirements are not too restric-

tive. Its best used in circumstances where values are sufficiently close together. It would be a useful substitute for sorting low value arrays where the grouping of values is very sufficiently close together, as such a big array where values are separated by the range of their values could be taken and have each subsection sorted and then combined. With this idea it would make the initial sorting of the separate arrays a near O(n) sort when the amount of values low enough, but merging of the sorted sub arrays could increase the overall time.

2.5 Patience sorting

2.5.1 The Concept

Just like tournament sorting this sorting algorithm is inspired by an already preexisting concept. Patience sorting is based off of the card game patience or its better known name of solitaire, which in the algorithmic case is used to efficiently calculate the length of a longest increasing sub-sequence in an array. Based of the game of the same name, the patience sort will being with a shuffled deck (unsorted array). After the shuffling the cards (elements) will be sorted into piles. The rules for creating piles is as follows the first card to be drawn will always become a new pile consisting of that first card. After that each subsequent card will be placed on the furthest left pile in which the top pile of card is of greater or equal value to the new card. If this condition is not met the new drawn card will be placed to the right making a new pile. The game will end once all cards have been dealt. Unlike the game the algorithm will not end here once all elements have been put into their respective piles the algorithm will start with a k-way merge of the piles (as each pile has already been sorted).

2.5.2 Pseudo code

```
1. procedure PatienceSort(array[]):
2.  for (i : array[]):
3.    if (i = 0):
4.     create a pile add array[1] to it
5.    else
6.      curPile = top element
           on leftmost pile
7.      if (currPile >= array[i]):
8         place array[i] on that pile
9.      else
10.       create a new pile to the right
           of the pile just compared
11. Merge all piles
```

2.5.3 Cost Analysis

In its based case when the elements with in the array have already been sorted it will result in the creation of a single pile as such its just going to be a O(n) cost as there is no need for a k-way merge. For both the worst and best case it will end up being O(n log n) as regardless of how close the pile are to being sorted they will still need to perform the k-way merge resulting the the cost being O(n log n).

2.5.4 Practical Application

Patience sorting has its application in process control in finding a sub sequence of increasing length is important. Finding such a sub sequence could point to an emerging trend marker. Overall any case where the need to find the existence/length of the longest increasing sub sequence is needed.

2.6 Stooge Sort

2.6.1 The Concept

This is a sorting algorithm that recursive in nature, overall the name stooge is meant to be a person that is a lackey or inferior in some way, just like this sorting algorithm. This sorting algorithm is notorious for being one of the worst sorting algorithms out there as its tie complexity is near O(n^3) complexity (O($n^{2.7095}$)). The stooge sort is meant to be an homage to the three stooges known for their slapstick comedy which usually involved these three actors usually inflicting harm on each other in a repetitive manner (like recursion). This sorting algorithm is fairly simple only having three distinct parts the first being checking if two array positions have their array elements swapped and then recursively sort the first two thirds and then the second two thirds, and then sort the first two thirds again. The reason for sorting the first two thirds again is to make sure that after sorting the second two thirds elements would have changed spots as such the middle one third could not be sorted as such the front two thirds needs to be sorted again.

2.6.2 Pseudo code

```
1. procedure stoogeSort(array[],i,last):
2.  if (array[last] < array[i]):
3.      swap(array[last], array[i])
4.
5.  if (last - i > 1):
6.      t = ((last-i)/3)
7.      stoogeSort(array,i,last-t)
8.      stoogeSort(array,i+t,last)
9.      stoogeSort(array,i,last-t)
```

2.6.3 Cost Analysis

Due to the nature of this array it will always try and recursively sort the first two thirds, the last two thirds, and the first two thirds again. With this in mind the run time of this algorithm will be 3T(3n/2) +1 which will give a cost of O($n^{log3/log1.5}$) or O($n^{2.7095}$).

2.6.4 Practical Application

Over all due to the terrible inefficient nature of this algorithm there is no real world use for this algorithm. There are no major changes that could be made to it

to allow for it to be come more efficient without losing what makes this algorithm a stooge sort. The only possible use for this is learning about the inefficient nature of this sort and to better understand how to not create inefficient sorting algorithms. Also hypothetically if you had database where it was constantly being added onto and it needed to be sorted even while new data was coming in then a stooge sort could allow for it to remain sorted at the front as with the nature of this hypothetical the data is being added to the end and as such the front could be sorted while the last two thirds would only remain sorted until new data is being inserted in. Of course this is only hypothetical and still this algorithm is very inefficient in nature.

2.7 References

Gnome Sort. GeeksforGeeks. (2021, March 31). https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/.

Guest, T. (2009). Word Aligned. Patience sort and the Longest increasing subsequence. http://wordaligned.org/articles/patience-sort.

Pancake sorting. GeeksforGeeks. (2020, October 22). https://www.geeksforgeeks.org/pancake-sorting/.

Rosetta Code. (n.d.). Sorting algorithms/Patience sort. Sorting algorithms/Patience sort - Rosetta Code. http://www.rosettacode.org/wiki/Sorting algorithms/Patience sort.

Sahai, H. (2019, June 10). Pigeonhole Sort. OpenGenus IQ: Learn Computer Science. https://iq.opengenus.org/pigeonhole-sorting/.

Stepanov, A., amp; Kershenbaum, A. (n.d.). Using Tournament Trees to Sort. Polytechnic Institute of New York. http://stepanovpapers.com/TournamentTrees.pdf.

Wikimedia Foundation. (2020, December 7). Stooge sort. Wikipedia. https://en.wikipedia.org/wiki/Stoogesort.

Wikimedia Foundation. (2021, January 26). Gnome sort.
Wikipedia. https://en.wikipedia.org/wiki/Gnome$_s ort$.