# CSE374: Final Project Report: Sorting Algorithms

Joe Alcini

May 2, 2021

## 1 Introduction

In 2005 Facebook CEO Mark Zuckerberg said during a lecture at Harvard that while scaling up Facebook user base to the size it is was growing that he faced many issues with sorting the most relevant connections that users shared between each other [11]. The issue was that it took a long time due to the massive amount of connections that exist and prioritizing which ones would be most relevant [11]. Facebook has only grown since that speech and now has over 2 billion registered users interacting with the platform on a monthly basis as shown by the graphic below [9]. So this may cause some to question how Facebook can sort through all of this data so quickly and effectively.

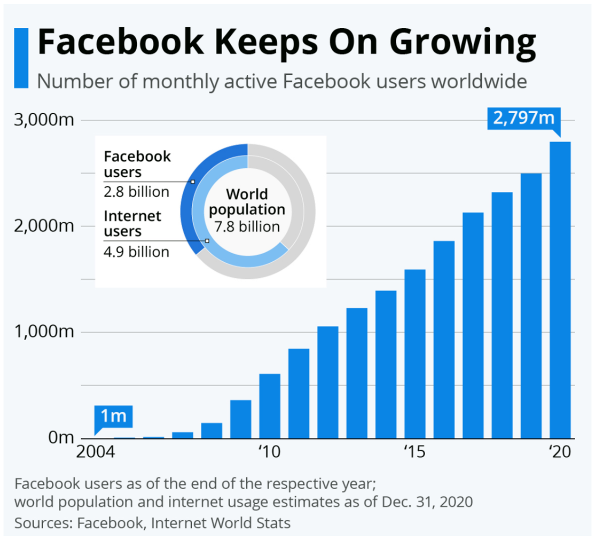


Figure 1: This graph shows the amount of Facebook Users over its life time. [9]

The previous example highlights one of the most common and prevalent challenges in Computer Science and Software Engineering is the need to quickly and efficiently sort a large amount of data for a predetermined task. While 10 to 20 objects can be sorted almost instantaneously by a computer, data-sets or other collections can contains hundreds of thousands or even millions of entries. This can lead to dramatic wait times for results, huge computing costs, and frustration by whomever is attempting to sort said data. To solve these issues various sorting algorithms, including quick sort and merge sort, have been designed in order to optimize sorting by reducing redundant calculations in reducing the time and resources required to solve large amounts of data.

Despite some of the well known algorithms listed above, this paper will attempt to dive into some more lesser used algorithms in order to judge their efficiency and determine whether or not they should be implemented in more software projects. The algorithms that will be examined and evaluated later on in this paper will include: Burnt Pancake Sorting, Stupid Sorting or Gnome Sort, Tournament Sorting, Kirkpatrick-Reisch Sorting, X + Y Sorting, and finally Patience Sorting.

These algorithms all fall into the categories of in place sorting , divide and conquer sorting, and heap sorting. In-place sorting works by sorting the data within the original list that was given. It does not create additional lists or other copies when sorting. Divide and Conquer strategies break the data up into many smaller lists until eventually they reach and small size, usually one element. Then these lists are compared against one another before combining them together until the original sized list exists again but in the sorted order. Due to the recursion of Divide and Conquer algorithms they are generally more efficient than a standard in-place sort. Heap Sorts work by adding element at a time to a tree like structure starting from the bottom and rearranging the nodes above it.

## 2 Sorting Algorithm Details

All of the algorithms will be given an overview of how they work, what type of algorithm they are classified as, a pseudo-code representation of how they function, an example of their usage, any special properties that the algorithm possesses, and finally how this algorithm can be practically applied to solve real problems that exist in computing.

### 2.1 Burnt Pancake sorting

#### 2.1.1 Concept

The Burnt Pancake sort gets its name because it is a variation of a basic pancake sort which utilizes replication flipping the order directly opposite of a set of data similar to flipping a stack of pancakes over. Just like flipping the pancakes when trying to flip over the stack you can put the spatula wherever you like within the stack however everything above it is going to change places [2]. The goal of said search is to order the elements of the data set so that the largest value is at the "bottom" of the data and the smallest is at the top. Likewise the "bottom" of each of the pancakes now becomes the top which is where the burnt pancake sort becomes begins to break from the original. This causes an additional check to make sure that the non burnt side is facing up so that however is seeing them wont suspect that they are burnt.

#### 2.1.2 Class

The Burnt Pancake Sort, like the standard, pancake sort is an in-place sorting algorithm. None of the flipping occurs anywhere but the stack that contains all of the values. This means that only one computation is occurring at a time that can lead to inefficient time to complete.

#### 2.1.3 Pseudo-Code

```
BURNT-PANCAKE-SORT(a)
        for i = a.length to 1
             max = FINDMAX(a, i-1)//find max index of the remaining values
             FLIP-STACK(max, i - 1)
             if a[max] < 0 //check if bottom side is down
                  a[max] = -1 * a[max]
             FLIP-STACK(a, a.length-i)

FIND-MAX(a, size)
        maxIndex = $-\infty$
        for i = 0 to size
             if abs val of a[maxIndex] ¡ abs val of a[i]
                  maxIndex = i
        return maxIndex

FLIP-STACK(max, size)
        for i = max to size / 2
             temp = a[i]
             a[i] = -1 * a[size - i]
             a[size - i] = -1 * temp
        return a
```

Where a is the array of values and size is the length of the unsorted array.

#### 2.1.4 Example

Pretend a chef has a stack of 5 pancakes that they just completed cooking. All of the pancakes have different diameters and all have one side that is burnt. The largest diameter should be at the bottom or furthest left of the list at the completion of the sort.

Consider the following diameters: [7, 3, 1, 9, 4] with 7 on the bottom and 4 on top. 9 and 3 have their brunt sides facing down.

They will be displayed as such: [7, -3, 1, -9, 4]

The first flip will occur after finding 9 as the largest resulting in this orientation: [7, -3, 1, -4, 9]

Finally 9 is sent to the bottom of the stack after another flip: [-9, 4, -1, 3, -7]

7 is now the largest remaining value but is bottom side is facing down it will be flipped up prior to flipping the remaining stack: [-9, 4, -1, 3, 7]

Now rotate the stack between 4 and 7: [-9, -7, -3, 1, -4]

4 now has the largest remaining absolute value of any pancake, while it is already at the top, it is facing the wrong way so it will be flipped: [-9, -7, -3, 1, 4]

Now flip the stack between -3 and 4: [-9, -7, -4, -1, 3]

3 is now the largest remaining and it is on top facing the right way so flip the stack between -1 and 3: [-9, -7, -4, -3, 1]

1 is now the largest and only remaining pancake in the stack orient it with the bottom side down to complete the sort: [-9, -7, -4, -3, -1]

#### 2.1.5 Time Complexity

The time complexity of the Burnt Pancake sort is represented by the time complexity of $O(n^2)$ where $n$ represents the amount of elements in the list. This is because two loops are needed to accurately order all of the entries that are fed through the algorithm. The first loop exists within the main part of the algorithm, BURNT-PANCAKE-SORT(a), and the second loop exists within the max value finder, FIND-MAX(a, size), or the flip, FLIP-STACK(max, size). Due to this large value for time complexity this algorithm is not generally used in practice because it is very slow for large data sets compared to other algorithms that are better optimized.

#### 2.1.6 Special Properties

A special property of this algorithm is that it changes the order of the elements in the list by mirroring their order regardless of the condition other than the one being solved. Another special property is that the sign changes for the elements that are flipped representing the other side is not facing up.

#### 2.1.7 Practical Applications

Despite the inefficiencies caused by the design of the Burnt Pancake Sort it has been used to solve some real world problems. One such example occurred when biologists used the burnt pancake sort in order to order the DNA of s specific strain of *E Coli*.

The result of this computation found that the "... cells would become antibiotic resistant when the segments are properly sorted." [7].

## 2.2 Stupid sorting (Gnome sort)

### 2.2.1 Concept

The next sorting algorithm that will be examined is the Stupid Sort, or also known as the gnome sort. The gnome sort, also known as stupid sorting, gets its name because one of the common explanations to help people understand it, originally coined by Dick Grune, uses the analogy of a garden gnome evaluating the comparisons between plants and making the swaps based on their size [5]. The sort works by comparing 2 items in a list and swapping two that are out of order. After fixing the first element out of order it will check the newly swapped element against all of the elements that have already been checked. This process continues until all elements are sorted.

### 2.2.2 Class

The gnome sort is an in place sorting Algorithm. The swaps only occur within the original list that contains all of the values. This also means that only one computation is being executed at a time and can lead to inefficient run times.

### 2.2.3 Pseudo-Code

```
STUPID-SORT(a)
    while index ¡ a.length // Loops until the end of the array
        if index == 0 // Checks starting index
            index = index + 1
        else if a[index] ≥ a[index - 1] // Checks the values of the indices
            index = index + 1
        else // If comparisons fail
            SWAP(a, index-1, index) // Changes the values
            index = index - 1

SWAP(a, index1, index2)
    int temp = a[index1]
    a[index1] = a[index2]
    a[index2] = a[index1]
```

Where a is the array of values and index is an index value in the unsorted array.

### 2.2.4 Example

Suppose the Gnome above has 5 plants that he plants to sort through of different heights. Assume the tallest plant should end up on the right.

Consider the following heights: [7, 14, 5, 19, 2]

| Description | Indices Compared | List |
|---|---|---|
| 7 is in correct position at start increase 1 index | 0 | [7, 14, 5, 19, 2] |
| 14 is in correct position increase 1 index | 0,1 | [7, 14, 5, 19, 2] |
| 5 is not in correct position swap with 14 decrease 1 index | 1,2 | [7, 5, 14, 19, 2] |
| 5 is not in correct position swap with 7 decrease 1 index | 0,1 | [5, 7, 14, 19, 2] |
| 5 is in correct position increase 1 index | 0 | [5, 7, 14, 19, 2] |
| 7 is in correct position increase 1 index | 0,1 | [5, 7, 14, 19, 2] |
| 14 is in correct position increase 1 index | 1,2 | [5, 7, 14, 19, 2] |
| 19 is in correct position increase 1 index | 2,3 | [5, 7, 14, 19, 2] |
| 2 is not in correct position swap with 19 decrease 1 index | 3,4 | [5, 7, 14, 2, 19] |
| 2 is not in correct position swap with 14 decrease 1 index | 2,3 | [5, 7, 2, 14, 19] |
| 2 is not in correct position swap with 7 decrease 1 index | 1,2 | [5, 2, 7, 14, 19] |
| 2 is not in correct position swap with57 decrease 1 index | 0,1 | [2, 5, 7, 14, 19] |
| 2 is in correct position increase 1 index | 0 | [2, 5, 7, 14, 19] |
| 5 is in correct position increase 1 index | 0,1 | [2, 5, 7, 14, 19] |
| 7 is in correct position increase 1 index | 1,2 | [2, 5, 7, 14, 19] |
| 14 is in correct position increase 1 index | 2,3 | [2, 5, 7, 14, 19] |
| 19 is in correct position and the end of the list | 3,4 | [2, 5, 7, 14, 19] |

The order of the plants by height shortest to tallest is: [2, 5, 7, 14, 19].

### 2.2.5 Time Complexity

The time complexity of the Stupid Sort is represented by the time complexity of $O(n^2)$ where $n$ represents the amount of elements in the list. Although it may appear at first glance to be less than $O(n^2)$ due to only one loop being present, increments and decrements exist. This causes many more than $n$ comparisons to be made and the large amounts of directional changes adds the effects of what an inner loop would to the time complexity overall making this a very inefficient algorithm and limits its uses in practice.

### 2.2.6 Special Properties

The special property of the Stupid Sort algorithm that differentiates it from other in place algorithms is its use of a single loop. The single loop can not only simplify the appearance of the code, but iterating through the list this way can reduce some unnecessary comparisons that other in place sorts make.

### 2.2.7 Practical Applications

Due to the inefficient running time of the Stupid Sort, it has not been applied in a professional scenario. Despite this the simplicity of the code structure makes it a great algorithm for students learning Computer Science and Algorithmic design to study and learn from.

## 2.3 Tournament sorting

### 2.3.1 Concept

The next algorithm that will be discussed is the Tournament Sort. The Tournament Sort gets its name because it creates a binary heap of a preset arbitrary length that replicates a bracket. New items are added to the bottom of the heap and compared with the parent node, or if none exists the sibling node. This continues until the heap is full and sorted in which the root node of the heap is removed and added to a new list. If a new item is smaller than the biggest item that has been previously removed then it can not be added to the removed list and blocks a space on the bottom most open node of the heap. Once the list runs out of elements or the heap is completely full the first iteration ends and two lists are formed from the removed nodes and the remaining nodes. These two lists are now set to one of the bottom nodes and the entries are compared one by one from each list until the heap is is either blocked again, in which the process repeats, or it is completely sorted, in which it terminates. [8]

### 2.3.2 Class

The Tournament Sort algorithm is a modification of the traditional Heap Sort algorithm and therefore they share many properties. The Tournament sort uses a tree like structure as is space to make the comparisons before they are pulled into a final list. The remaining lists that still need compared also follow this comparison measure until completion.

### 2.3.3 Pseudo-Code

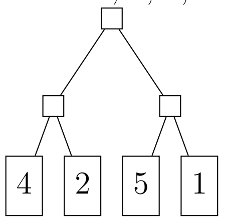
```
TOURNAMENT-SORT(a)
        sorted = $\emptyset$
        while a is not $\emptyset$
             tree = CREATE-TREE(k) // Creates binary tree depth k, all = $-\infty$
             // Adds elements to the bottom of the tree
             ADD-FIRST-ELEMENTS-TO-TREE(a[0 to tree.bottom.length])
             MATCH-UPS(tree, a, sorted)
             sorted,a = COMBINE(a, sorted)
```

```
MATCH-UPS(tree, a, sorted)
        while tree !empty or root is invalid // Invalid vals cant be added to sorted
             if root !null
                  add root to sorted
             if node children !null
                  if child1 ¡ child2: node = child1
                  else: node = child2
             if node children null and not > sorted[sorted.length]
                  child = a[0]
             else:
                  child = a[0] and child is in valid
        a = EMPTY-TREE(tree) + a
        return sorted, a

COMBINE(a, sorted)      newSorted = ∅
        while a not or sorted not empty
             if a[0] ¡ sorted[0]: add a[0] to newSorted
             else: add sorted[0] to newSorted
        if a is empty: add remaining sorted to newSorted
        return newSorted, a
```

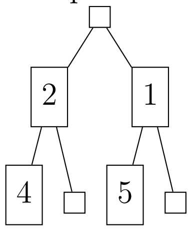
### 2.3.4 Example

For this example imagine we have a tournament field of baseball teams ordered by the amount of wins that they have on the year in ascending order. However a data scientist wants to order the teams by the average amount of runs a tam scores in a game in ascending order. The starting list of teams runs is as follows: [4, 2, 5, 1, 7, 9, 6, 5]

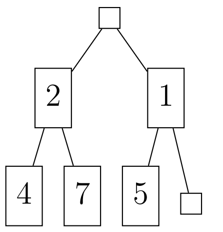
Insert 4, 2, 5, 1 into the bottom row



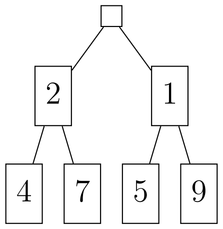
Compare 4 and 2, and 5 and 1 and move the min values up



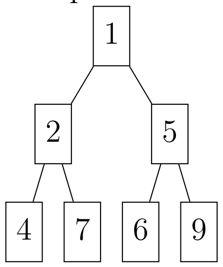
Insert 7 and compare with 2



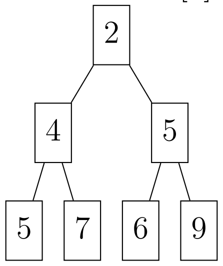
Insert 9 and compare with 1



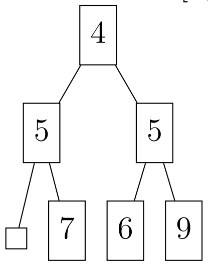
Compare 1 and 2 and nodes below 1, Insert 6



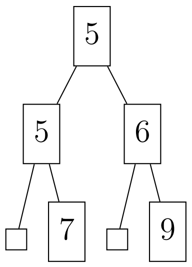
Pull 1 add it to the final list, Compare 2 and 5, Add 5
Sorted List: [1]



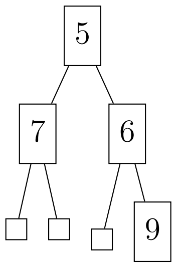
Pull 2 add it to final list, compare 4 and 5
Sorted List: [1, 2]



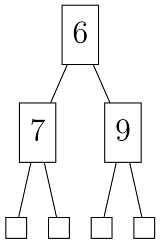
Pull 4 add it to final list, compare 5 and 5
Sorted List: [1, 2, 4]



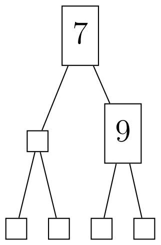
Pull 5 add it to final list, compare 5 and 6
Sorted List: [1, 2, 4, 5]



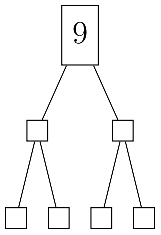
Pull 5 add it to final list, compare 7 and 6
Sorted List: [1, 2, 4, 5, 5]



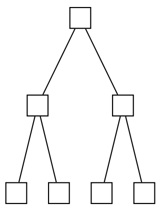
Pull 6 add it to final list, compare 7 and 9
Sorted List: [1, 2, 4, 5, 5, 6]



Pull 7 add it to final list, push 9 to root
Sorted List: [1, 2, 4, 5, 5, 6, 7]



Pull 9 add it to the final list to complete the sort
Sorted List: [1, 2, 4, 5, 5, 6, 7, 9]



Resulting sorted list: [1, 2, 4, 5, 5, 6, 7, 9]

### 2.3.5 Time Complexity

The time complexity of the Tournament Sort is represented by the time complexity of $O(nlog(n))$ where $n$ represents the amount of elements in the list. This is a result of the heap taking an a initial time of $O(n)$ to create the beginning heap and makes $O(log(n))$ comparisons after creating the original heap.

### 2.3.6 Special Properties

The tournament sort uses a heap of a set size to make the comparisons instead of building a bigger one each time. It also creates multiple lists that it uses for multiple iterations of comparisons once that heap size is reached due to out of order values.

### 2.3.7 Practical Applications

Due to the efficient run time of the Tournament Sort it can be used in similar situations that a Heap Sort would be used such as the combination of two lists of data which requires a reliable time estimate to compute such as processing and ordering data of a rocket's telemetry to calculate what it should do next.

## 2.4 Kirkpatrick–Reisch Sort

### 2.4.1 Concept

The next algorithm we will look at Kirkpatrick–Reisch Sort is an algorithm that resembles a similar appearance to Huffman Encoding. This algorithm uses a tree and splitting significant bits from less significant bits where each original entry is given a branch. If multiple entries have the same starting bits as another the remaining bits become children that are later sorted and recombines. The significant bit nodes are then compared against each other and the smallest goes on the left and the largest on the right and then the child nodes are ordered in the same manner. Finally, similar to calculating the values of hash tables the tree is read from left to right while travelling down the branches top to bottom.

### 2.4.2 Class

The Kirkpatrick–Reisch Sort works similar to how a heap is sorted and also utilizes a recursive element [3]. The tree uses a method of taking the largest values and making them their own layer of the tree before organizing the child nodes like a heap. Breadth First Traversals are also used to sort the most significant bits by size but keeping their children with them. The items are finally computed starting from the main node and adding the value on the path at every level to the back of the number until it reaches the end.

### 2.4.3 Pseudo-Code

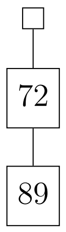
```
Kirkpatrick–Reisch-Sort(a, w)
        tree = ∅
        sorted = ∅
        for i=0 in a.length
            tree = ADD(a[i], w) // Adds to the tree and splits it
        tree = REORDER(tree)
        for i=0 in a.length
            sorted = sorted + Remove(tree) // Takes small val and adds to sorted

REORDER(tree)
        if tree does not have child
            return tree
            if tree has child
                REORDER(tree)
                BREADTH-FIRST-SEARCH()
    return tree
```

### 2.4.4 Example

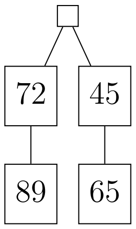
Consider a series of hash values come out of order and we would like to order them. We will a tree with 2 level below the root in order to sort them.

Consider this list of hash values: [7289, 4567, 7234, 1101, 1167]

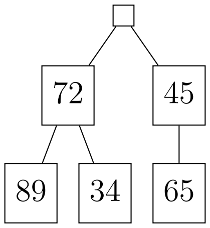
Add 7289 to the tree splitting it in 2 parts



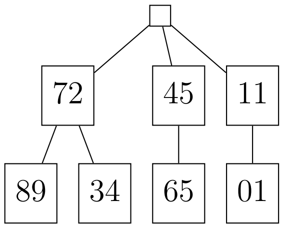
Add 4567 to the tree splitting it in 2 parts



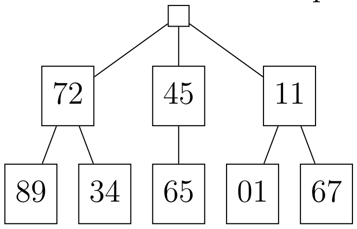
Add 7234 to the tree splitting it in 2 parts



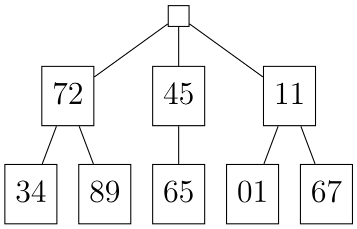
Add 1101 to the tree splitting it in 2 parts



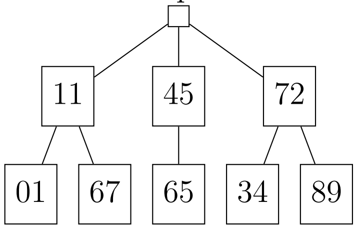
Add 1167 to the tree splitting it in 2 parts



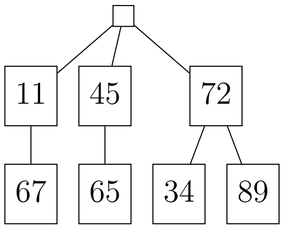
Reorder the leaf nodes under their parents



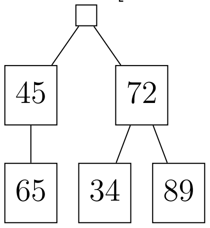
Reorder the parent nodes under the root node



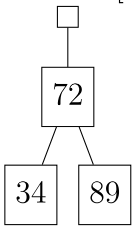
Pull 1101 and add to the sorted list
Sorted: [1101]



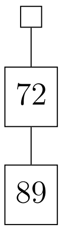
Pull 1167 and add to the sorted list. Remove 11 since no more children
Sorted: [1101, 1167]



Pull 4565 and add to the sorted list. Remove 45 since no more children
Sorted: [1101, 1167, 4565]



Pull 7234 and add to the sorted list
Sorted: [1101, 1167, 4565, 7234]



Pull 7289 and add to the sorted list
Sorted: [1101, 1167, 4565, 7234, 7289]

No more nodes exist the sort is complete
Result: [1101, 1167, 4565, 7234, 7289]

### 2.4.5 Time Complexity

The complexity of the Kirkpatrick–Reisch Sort is $O(nlog(n))$. This is due to the original list having $n$items taking $O(n)$ to join. Also because of the recursive properties of the sort it takes $O(log(n))$ to recombine all of the pieces of each entry.

### 2.4.6 Special Properties

The Kirkpatrick-Reisch sort breaks down individual entries into smaller components. This makes it possible to sort multiple indices at once with out as many comparisons.

### 2.4.7 Practical Applications

The Kirkpatrick-Reisch sort can work well if it is used in tandem with a hashing function. The tree design and splits and data can make it an efficient way to solve collisions. [3]

## 2.5 Patience sorting

### 2.5.1 Concept

The Patience Sort is named for its similarity to the optimal solution for a card game that goes by the same name. The patience sort attempts to find the largest amount of descending values as possible and group them together and then sort from the smallest of each of the values and then trying to combine the smaller groups into the sorted big group [6].

### 2.5.2 Class

The Patience Sort is most often compared with Quick Sort due to the sub groups it creates. The Quick Sort uses partitions to create smaller sub problems of unequal length similar to the patience sort. However, the Patience Sort will divide into subsections using a greedy process of selecting the smallest top value of all the existing stacks that is smaller than the new value and adds it to it. The sort also has a merging step to combine the stacks but these only combine one element of the lists at a time.

### 2.5.3 Pseudo-Code

```
PATIENCE-Sort(a)
        Stacks = $\emptyset$ // Creates a priority queue order by last element
        Sorted = $\emptyset$
        Stacks = CREATE-STACKS(a, Stacks)
        while Stacks not empty
            for Stack in Stacks
                find min value
            add min to Sorted
        **return** Sorted
CREATE-STACKS(a, Stacks)
        for $i = 0$ to a.length
            if Stacks.size == 0 // First element in array
                create new Stack in Stacks
                add a[i] to stack
            else
                while stack in Stacks
                    if Stack.lastNumber ¿ a[i]
                        add a[i] to stack
                if not less than any stack min
                    create new Stack in Stacks
                    add a[i] to the new Stack
```

### 2.5.4 Example

Suppose we have a stack of unordered objects numbered one to ten. They initially appear in the following order: [4, 7, 3, 8, 9, 2, 1, 5, 6, 10]. Sort the list using a patience sort to place the list of objects in ascending order.

Start by creating the smaller piles:

| Description | Original | Sorting Piles |
|---|---|---|
| Create a new pile with 4 | [7, 3, 8, 9, 2, 1, 5, 6, 10] | [[4]] |
| Create a new pile with 7 | [3, 8, 9, 2, 1, 5, 6, 10] | [[4],[7]] |
| Add 3 to the pile with 4 | [8, 9, 2, 1, 5, 6, 10] | [[3,4],[7]] |
| Create a new pile with 8 | [9, 2, 1, 5, 6, 10] | [[3,4],[7],[8]] |
| Create a new pile with 9 | [2, 1, 5, 6, 10] | [[3,4],[7],[8],[9]] |
| Add 2 to the pile with 3 | [1, 5, 6, 10] | [[2,3,4],[7],[8],[9]] |
| Add 1 to the pile with 2 | [5, 6, 10] | [[1,2,3,4],[7],[8],[9]] |
| Add 5 to the pile with 7 | [6, 10] | [[1,2,3,4],[5,7],[8],[9]] |
| Add 6 to the pile with 8 | [10] | [[1,2,3,4],[5,7],[6,8],[9]] |
| Create a new pile with 10 | [] | [[1,2,3,4],[5,7],[6,8],[9],[10]] |

Remove the smallest overall value from the piles and add it to the sorted list

| Description | Sorting Piles | Sorted |
|---|---|---|
| Remove 1 | [[2,3,4],[5,7],[6,8],[9],[10]] | [1] |
| Remove 2 | [[3,4],[5,7],[6,8],[9],[10]] | [1,2] |
| Remove 3 | [[4],[5,7],[6,8],[9],[10]] | [1,2,3] |
| Remove 4 | [[5,7],[6,8],[9],[10]] | [1,2,3,4] |
| Remove 5 | [[7],[6,8],[9],[10]] | [1,2,3,4,5] |
| Remove 6 | [[7],[8],[9],[10]] | [1,2,3,4,5,6] |
| Remove 7 | [[8],[9],[10]] | [1,2,3,4,5,6,7] |
| Remove 8 | [[9],[10]] | [1,2,3,4,5,6,7,8] |
| Remove 9 | [[10]] | [1,2,3,4,5,6,7,8,9] |
| Remove 10 | [] | [1,2,3,4,5,6,7,8,9,10] |

The result of the sorted list is: [1,2,3,4,5,6,7,8,9,10]

#### 2.5.5 Time Complexity

The complexity of the Patience Sort is $O(nlog(n))$. This is due to the initial loop through all of the values in the list at least once and making the subgroups in $O(n)$ time. The $O(log(n))$ part is derived from the combinations of the sub groups that reduce the overall components that need to be checked.

#### 2.5.6 Special Properties

A special property of the Patience sort is the creation of the subgroups. The subgroups don't have a preset limit of how many can be created at a time but can be created to have up to $n$ piles if the list is in reverse order.

#### 2.5.7 Practical Applications

The patience sort has been used to calculate the longest increasing sub-sequence in an unsorted list of numbers. This was proven by Aldous and Diaconis and used a series of pointers to the most recent number added to the sub groups and then following the series of pointers backwards until it no longer works. [1]

### 2.6 X + Y sorting

#### 2.6.1 Concept

The final algorithm that will be examined is the X + Y sort. The X + Y sort is different from the other algorithms that are covered because it involves creating multiple new values in its sorting process. The new values are created b creating a sum of every number that exists in two lists of elements called X and Y. The resulting sums are then ordered to find the lowest or highest combination of values.

#### 2.6.2 Class

The X + Y sort is a theoretical algorithm that has recursive elements. The recursive elements exist when trying to sort the combinations of inputs using a proven sorting method such as the merge sort. This occurs three times once to sort each list and then once to sort the combinations of the elements in both lists.

#### 2.6.3 Pseudo-Code

```
X+Y-SORT(X, Y)
    sums = $\emptyset$
    MERGE-SORT(X)
    MERGE-SORT(Y)
    for i in X.length
```

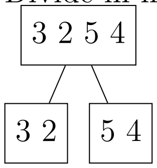
```
        for j in Y.length
            add X[i] + Y[j] to sums
    MERGE-SORT(sums)
```

### 2.6.4 Example

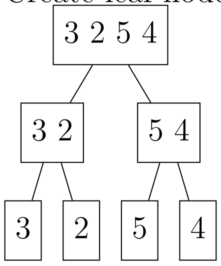
Take two lists of bus ticket prices X and Y that are needed to reach a destination. The two busses must be taken in order to reach the intended destination. List X has prices: [3, 2, 5, 4], and list Y has prices: [4, 6, 2, 7]. Find the cheapest price of the bus tickets.

Sort List X:

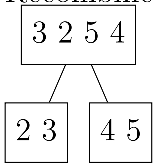
Divide in half



Create leaf nodes



Recombine leaves

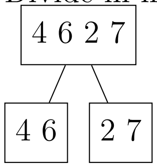


Recombine Halves

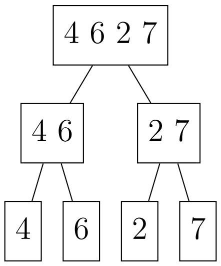
$\boxed{2\ 3\ 4\ 5}$

Sort List Y:

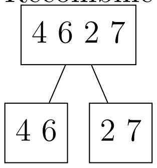
Divide in half



Create leaf nodes



Recombine leaves



Recombine halves

$\boxed{2\ 4\ 6\ 7}$

Create Sums:

Create Sums for X value 2

| **X/Y** | **2** | **4** | **6** | **7** |
|---|---|---|---|---|
| **2** | 4 | 6 | 8 | 9 |
| **3** | | | | |
| **4** | | | | |
| **5** | | | | |

The list of sums is: [4, 6, 8, 9]

Create Sums for X value 3

| **X/Y** | **2** | **4** | **6** | **7** |
|---|---|---|---|---|
| **2** | 4 | 6 | 8 | 9 |
| **3** | 5 | 7 | 9 | 10 |
| **4** | | | | |
| **5** | | | | |

The list of sums is: [4, 6, 8, 9, 5, 7, 9, 10]

Create Sums for X value 4

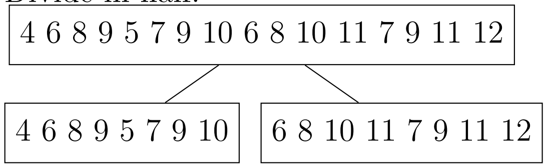
| **X/Y** | **2** | **4** | **6** | **7** |
|---|---|---|---|---|
| **2** | 4 | 6 | 8 | 9 |
| **3** | 5 | 7 | 9 | 10 |
| **4** | 6 | 8 | 10 | 11 |
| **5** | | | | |

The list of sums is: [4, 6, 8, 9, 5, 7, 9, 10, 6, 8, 10, 11]

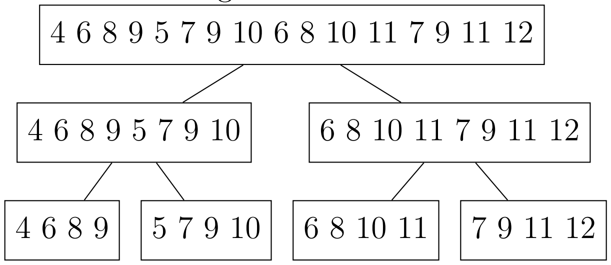
Create Sums for X value 5

| **X/Y** | **2** | **4** | **6** | **7** |
|---|---|---|---|---|
| **2** | 4 | 6 | 8 | 9 |
| **3** | 5 | 7 | 9 | 10 |
| **4** | 6 | 8 | 10 | 11 |
| **5** | 7 | 9 | 11 | 12 |

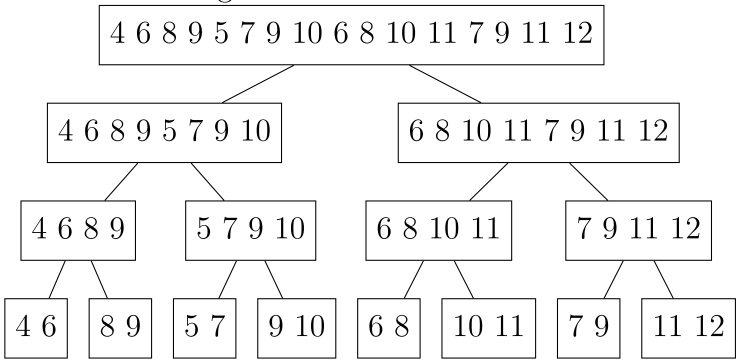
The list of sums is: [4, 6, 8, 9, 5, 7, 9, 10, 6, 8, 10, 11, 7, 9, 11, 12]
Sort the list of sums using merge sort:
Divide in half:



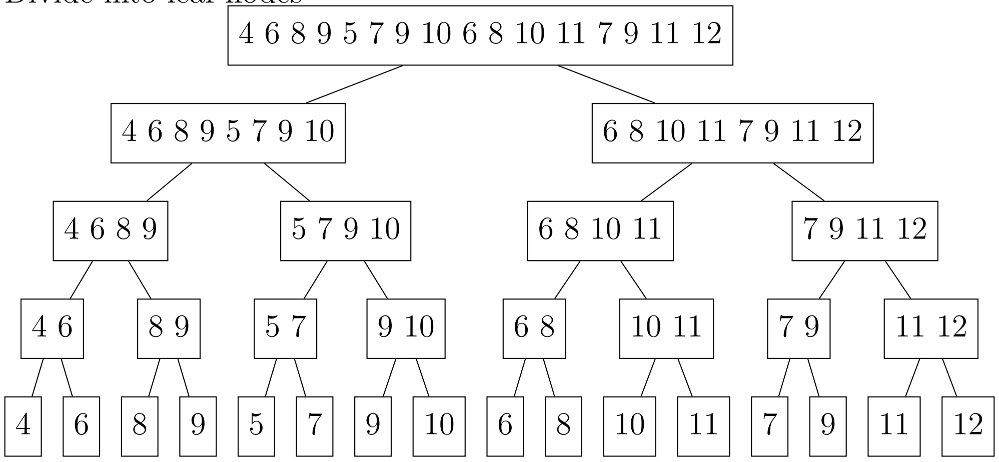
Divide in half again



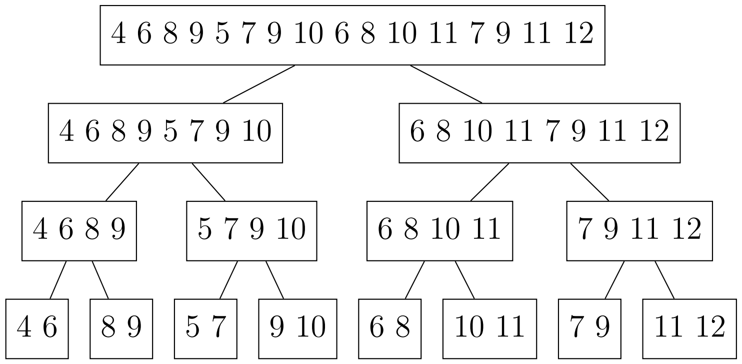
Divide in half again



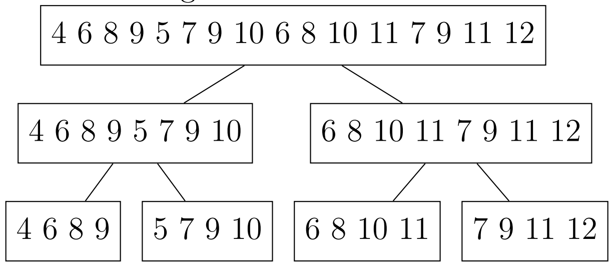
Divide into leaf nodes



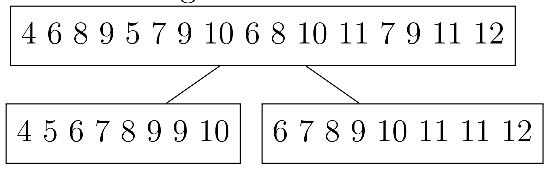
Recombine



Recombine again



Recombine again



Recombine the halves
Recombine again

$\boxed{4\ 5\ 6\ 6\ 7\ 7\ 8\ 8\ 9\ 9\ 9\ 10\ 10\ 11\ 11\ 12}$

This list of sorted bus ticket combos is [4, 5, 6, 6, 7, 7, 8, 8, 9, 9, 9, 10, 10, 11, 11, 12]

### 2.6.5 Time Complexity

The time complexity of the X + Y sort is $O(n^2 log(n))$. This is due to the creation of $n$ sums for each element in a list of $n$ length resulting in an initial complexity of $O(n^2)$. The $O(logn(n))$ comes from using a merge sort to sort the sums of numbers.

### 2.6.6 Special Properties

The X + Y sort is unique because it combines the values and orders them by their sums. This creates many extra values and comparisons that are not seen in other sorting algorithms.

### 2.6.7 Practical Applications

The X + Y sort mainly remains as a theoretical algorithm without any true real world uses. Author Steven Skiena has theorized that this algorithm could be applied to a system that could compare the prices of different transportation methods in order to find the cheapest total costs. [10]

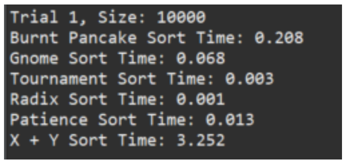
# 3 Results and Conclusions

## 3.1 Implementation and Results

Now that all of the algorithms and their theoretical time complexities have been introduced it is now time to test their actual running time efficiencies. To test these algorithms each one will be developed with the exception of the Kirkpatrick-Reisch sort which will be substituted with the Radix sort. The Radix sort is chosen because it behaves similar to the Kirkpatrick-Resich sort because both deal with sorting individual digits of the values [4].

The tests that will occur will create a random list of integers of size 10, 50, 100, 500, 1000, 5000, and 10000. Each algorithm will be fed a copy of the unsorted list and the time that it takes to solve each will be measured and compared. Five tests at each level will be taken and averaged in order to find the average running time of each algorithm at each level.

An example of what a user would see if they performed the test is shown below and represents a console output that includes the trial number of the size of the input array and the sorting time in seconds it took each algorithm to complete the sort.



```
Trial 1, Size: 10000
Burnt Pancake Sort Time: 0.208
Gnome Sort Time: 0.068
Tournament Sort Time: 0.003
Radix Sort Time: 0.001
Patience Sort Time: 0.013
X + Y Sort Time: 3.252
```

Figure 2: Shows the output of one trial of the tests

The full results of each trial of the test are shown below and the units are measured in seconds. The headings for each column will be abbreviated in order to have enough space to fit the results. BPS is the Burnt Pancake Sort, GS is the Gnome Sort, TS is the Tournament Sort, RS is the Radix Sort that is replacing the Kirkpatrick-Reisch Sort, PS is the Patience Sort, and X+Y is the X+Y Sort.

The results for the 10 element tests are:

| $n = 10$ | **BPS** | **GS** | **TS** | **RS** | **PS** | **X+Y** |
|---|---|---|---|---|---|---|
| **Trial 1** | 0.0005 | 0.0005 | 0.001 | 0.001 | 0.002 | 0.001 |
| **Trial 2** | 0.0005 | 0.0005 | 0.0005 | 0.001 | 0.001 | 0.001 |
| **Trial 3** | 0.0005 | 0.0005 | 0.001 | 0.001 | 0.001 | 0.0005 |
| **Trial 4** | 0.0005 | 0.0005 | 0.001 | 0.001 | 0.001 | 0.001 |
| **Trial 5** | 0.0005 | 0.0005 | 0.001 | 0.001 | 0.001 | 0.0005 |
| **Mean** | **0.0005** | **0.0005** | **0.0009** | **0.001** | **0.0012** | **0.0008** |

The results for the 10 element array are very similar because the lack of elements leads to less variation and swaps to occur in some of the inefficient algorithms. The Burnt Pancake sort and Gnome Sort both had average times sorting times of 0.0005 seconds and were the most efficient. The next most efficient algorithms were the X+Y Sort with a sorting time of 0.0008 seconds, the Tournament Sort with a sorting time of 0.0009 seconds, and the Radix Sort with a sorting time of 0.001 seconds. Finally the Patience Sort had the most inefficient sorting time which took 0.0012 seconds, and the longest individual time in any trial at 0.002 seconds.

The results for the 50 element tests are:

| $n = 50$ | **BPS** | **GS** | **TS** | **RS** | **PS** | **X+Y** |
|---|---|---|---|---|---|---|
| **Trial 1** | 0.0005 | 0.0005 | 0.001 | 0.001 | 0.002 | 0.002 |
| **Trial 2** | 0.0005 | 0.0005 | 0.001 | 0.0005 | 0.002 | 0.001 |
| **Trial 3** | 0.0005 | 0.0005 | 0.001 | 0.0005 | 0.002 | 0.001 |
| **Trial 4** | 0.0005 | 0.0005 | 0.001 | 0.0005 | 0.001 | 0.002 |
| **Trial 5** | 0.0005 | 0.0005 | 0.001 | 0.001 | 0.001 | 0.001 |
| **Mean** | **0.0005** | **0.0005** | **0.001** | **0.0007** | **0.0018** | **0.0012** |

The next size of the unsorted array was 50 elements. The results are very similar to the 10 element array because the lack of elements leads to less variation and swaps to occur in some of the inefficient algorithms. The Burnt Pancake Sort and Gnome Sort both had average times sorting times of 0.0005 seconds and were the most efficient. The next most efficient algorithms were the Radix sort with a sorting time of 0.0007 seconds, Tournament Sort with a sorting time of 0.001 seconds, the X+Y sort with a sorting time of 0.0012 seconds. Finally, the Patience Sort had the most inefficient sorting time which took 0.0018 seconds.

The results for the 100 element tests are:

| $n = 100$ | **BPS** | **GS** | **TS** | **RS** | **PS** | **X+Y** |
|---|---|---|---|---|---|---|
| **Trial 1** | 0.001 | 0.0005 | 0.001 | 0.001 | 0.002 | 0.002 |
| **Trial 2** | 0.001 | 0.0005 | 0.002 | 0.0005 | 0.002 | 0.002 |
| **Trial 3** | 0.001 | 0.0005 | 0.001 | 0.0005 | 0.002 | 0.002 |
| **Trial 4** | 0.001 | 0.0005 | 0.001 | 0.001 | 0.002 | 0.003 |
| **Trial 5** | 0.0005 | 0.002 | 0.001 | 0.001 | 0.002 | 0.002 |
| **Mean** | **0.0009** | **0.0008** | **0.0012** | **0.0008** | **0.002** | **0.0022** |

The next size of the unsorted array was 100 elements. This size led to results that were similar to the 50 element array with some slight variations that begin to reflect the expected times it would take to sort the lists. The Gnome Sort and Radix Sort were the most efficient algorithms and had a sorting time of 0.0008 seconds. The next most efficient algorithms were the Burnt Pancake Sort with a sorting time of 0.0009 seconds, the Tournament Sort with a sorting time of 0.0012 seconds, and the Patience Sort with a sorting time of 0.002 seconds. Finally, the X+Y Sort had the most inefficient sorting time which took 0.0022 seconds.

The results for the 500 element tests are:

| $n = 500$ | **BPS** | **GS** | **TS** | **RS** | **PS** | **X+Y** |
|---|---|---|---|---|---|---|
| **Trial 1** | 0.005 | 0.003 | 0.002 | 0.001 | 0.004 | 0.022 |
| **Trial 2** | 0.004 | 0.004 | 0.001 | 0.001 | 0.005 | 0.022 |
| **Trial 3** | 0.003 | 0.004 | 0.004 | 0.001 | 0.009 | 0.02 |
| **Trial 4** | 0.004 | 0.003 | 0.002 | 0.001 | 0.003 | 0.017 |
| **Trial 5** | 0.004 | 0.003 | 0.002 | 0.001 | 0.004 | 0.019 |
| **Mean** | **0.004** | **0.0034** | **0.0022** | **0.001** | **0.005** | **0.02** |

The next size of the unsorted array was 500 elements. This input size was a very good example of how inefficiently designed algorithms can lead to poorer results than other algorithms that have a better design. The most efficient algorithm was the Radix Sort with a sorting time of 0.001 seconds. The next most efficient algorithms were the Tournament Sort with a sorting time of 0.0022 seconds, the Gnome Sort with a sorting time of 0.0034 seconds, the Burnt Pancake Sort with a sorting time of 0.004 seconds, and the Patience Sort with a sorting time of 0.005 seconds. Finally, the X+Y Sort had the most inefficient sorting time which took 0.02 seconds.

The results for the 1000 element tests are:

| $n = 1000$ | **BPS** | **GS** | **TS** | **RS** | **PS** | **X+Y** |
|---|---|---|---|---|---|---|
| **Trial 1** | 0.012 | 0.006 | 0.002 | 0.001 | 0.007 | 0.062 |
| **Trial 2** | 0.011 | 0.005 | 0.002 | 0.001 | 0.008 | 0.059 |
| **Trial 3** | 0.011 | 0.005 | 0.002 | 0.001 | 0.01 | 0.062 |
| **Trial 4** | 0.01 | 0.005 | 0.003 | 0.001 | 0.009 | 0.054 |
| **Trial 5** | 0.018 | 0.005 | 0.002 | 0.001 | 0.008 | 0.06 |
| **Mean** | **0.0124** | **0.0052** | **0.0022** | **0.001** | **0.0084** | **0.0594** |

The next size of the unsorted array was 1000 elements. This input size also reinforced how inefficiently designed algorithms can lead to poorer results than other algorithms that have a better design. The most efficient algorithm was the Radix Sort with a sorting time of 0.001 seconds. The next most efficient algorithms were the Tournament Sort with a sorting time of 0.0022 seconds, the Gnome Sort with a sorting time of 0.0052 seconds, the Patience Sort with a sorting time of 0.0084 seconds, and the Burnt Pancake Sort with a sorting time of 0.0124 seconds. Finally, the X+Y Sort had the most inefficient sorting time which took 0.0594 seconds.

The results for the 5000 element tests are:

| $n = 5000$ | **BPS** | **GS** | **TS** | **RS** | **PS** | **X+Y** |
|---|---|---|---|---|---|---|
| **Trial 1** | 0.073 | 0.03 | 0.007 | 0.003 | 0.018 | 1.324 |
| **Trial 2** | 0.066 | 0.022 | 0.006 | 0.002 | 0.017 | 1.321 |
| **Trial 3** | 0.067 | 0.028 | 0.006 | 0.002 | 0.018 | 1.306 |
| **Trial 4** | 0.073 | 0.036 | 0.007 | 0.004 | 0.028 | 1.429 |
| **Trial 5** | 0.073 | 0.034 | 0.012 | 0.006 | 0.026 | 1.193 |
| **Mean** | **0.0704** | **0.03** | **0.0076** | **0.0034** | **0.0214** | **1.3146** |

The next input size of the unsorted array was 5000 elements. This input size began to reveal massive differences in the amount of time it took to sort the arrays with the various sorting algorithms. The most efficient algorithm was the Radix Sort with a sorting time of 0.0034 seconds. The next most efficient algorithms were the Tournament Sort with a sorting time of 0.0076 seconds, the Patience Sort with a sorting time of 0.0214 seconds, the Gnome Sort with a sorting time of 0.03 seconds, and the Burnt Pancake Sort with a sorting time of 0.0704 seconds. Finally, the X+Y Sort had the most inefficient sorting time which took 1.3146 seconds.

The results for the 10000 element tests are:

| $n = 10000$ | **BPS** | **GS** | **TS** | **RS** | **PS** | **X+Y** |
|---|---|---|---|---|---|---|
| **Trial 1** | 0.285 | 0.103 | 0.014 | 0.005 | 0.034 | 3.793 |
| **Trial 2** | 0.257 | 0.095 | 0.012 | 0.004 | 0.026 | 3.715 |
| **Trial 3** | 0.271 | 0.091 | 0.016 | 0.004 | 0.028 | 3.963 |
| **Trial 4** | 0.249 | 0.114 | 0.011 | 0.005 | 0.027 | 3.901 |
| **Trial 5** | 0.257 | 0.09 | 0.007 | 0.005 | 0.031 | 3.972 |
| **Mean** | **0.2638** | **0.0986** | **0.012** | **0.0046** | **0.0292** | **3.8688** |

The next and final input size of the unsorted array was 10000 elements. This input size fully revealed the massive differences in the amount of time it took to sort the arrays with the various sorting algorithms. The most efficient algorithm was the Radix Sort with a sorting time of 0.0046 seconds. The next most efficient algorithms were the Tournament Sort with a sorting time of 0.012 seconds, the Patience Sort with a sorting time of 0.0292 seconds, the Gnome Sort with a sorting time of 0.0986 seconds, and the Burnt Pancake Sort with a sorting time of 0.2638 seconds. Finally, the X+Y Sort had the most inefficient sorting time which took 3.8688 seconds.

The averages of every test are as follows:

| | **BPS** | **GS** | **TS** | **RS** | **PS** | **X+Y** |
|---|---|---|---|---|---|---|
| $n = 10$ | 0.0005 | 0.0005 | 0.0009 | 0.001 | 0.0012 | 0.0008 |
| $n = 50$ | 0.0005 | 0.0005 | 0.001 | 0.0007 | 0.0018 | 0.0012 |
| $n = 100$ | 0.0009 | 0.0008 | 0.0012 | 0.0008 | 0.002 | 0.0022 |
| $n = 500$ | 0.004 | 0.0034 | 0.0022 | 0.001 | 0.005 | 0.02 |
| $n = 1000$ | 0.0124 | 0.0052 | 0.0022 | 0.001 | 0.0084 | 0.0594 |
| $n = 5000$ | 0.0704 | 0.03 | 0.0076 | 0.0034 | 0.0214 | 1.3146 |
| $n = 10000$ | 0.2638 | 0.0986 | 0.012 | 0.0046 | 0.0292 | 3.8688 |

The times it takes to sort the data increases every time as the size gets bigger for all algorithms. This is as expected along with the rates at which the times changed. The algorithms with higher theoretical time complexities often were the search algorithms that to the longest to sort and the ones that have lower theoretical time complexities were often the quickest to sort.

The following graphs will attempt to visually display the results of the trials and will have descriptions to help highlight some of the trends.

This graph displays the sorting times of all of the algorithms that were tested as a line for each. The X+Y sort is the focal point of this graph because it has many more values to compare than the other sorts at the same input size and as a result takes a much longer time to sort than the other algorithms.

This graph displays the times of the sorting algorithms without the X+Y Sort to give a better view of how the other algorithms performed against the random data. The Burnt Pancake Sort had the longest running time to sort and Gnome Sort had the

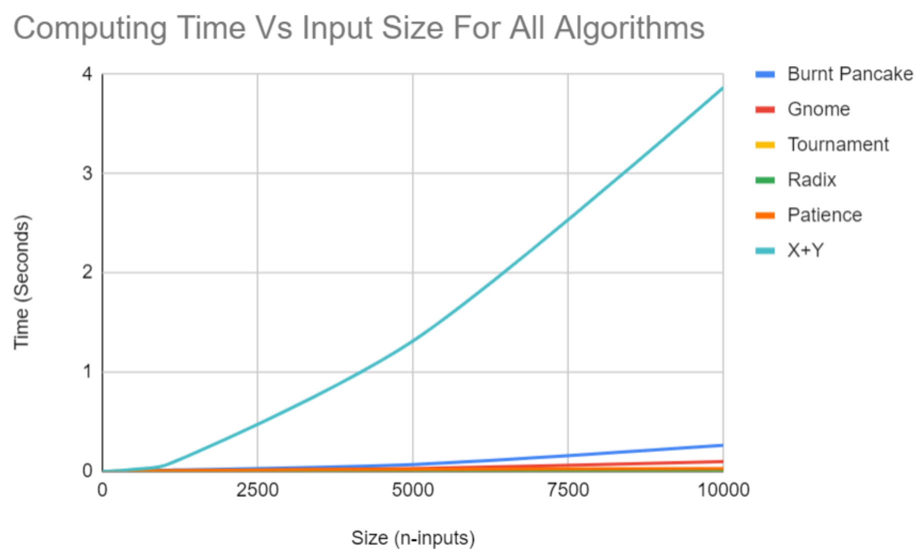


Figure 3: Shows a line for each algorithm which represents the average time it takes to sort each input size

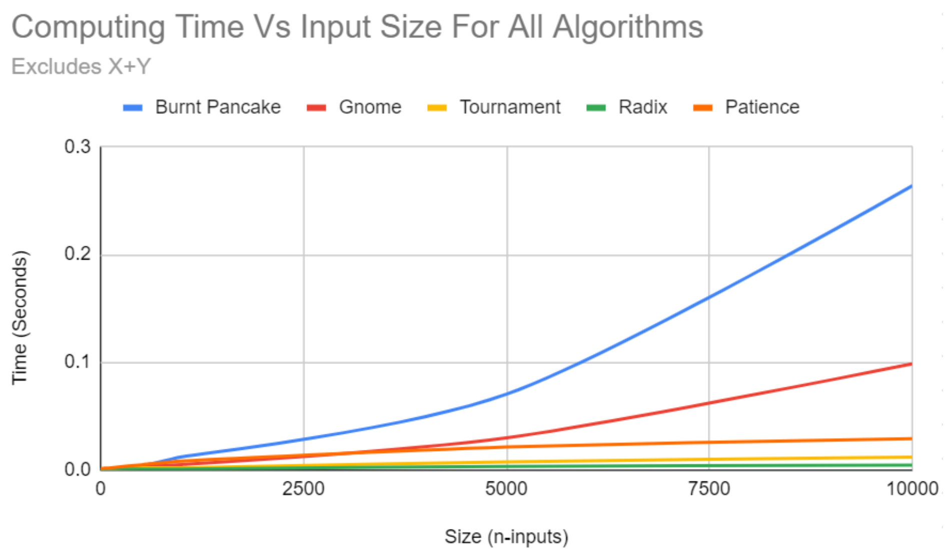


Figure 4: Shows a line for each algorithm which represents the average time it takes to sort each input size. Does not include the X+Y Sort in order to make other algorithms visible

second longest running time. This is consistent with the time complexities covered for each algorithm in the earlier sections. Both of these algorithms had a time complexity of $O(n^2)$. The other algorithms displayed on this graph have a complexity the is lower than $O(n^2)$. The differences in times between the Burnt Pancake Sort and Gnome Sort cold be derived from implementation errors in creating the most efficient version of each sort or from the Burnt Pancake sort having to make unnecessary comparisons due to the flips that occur.

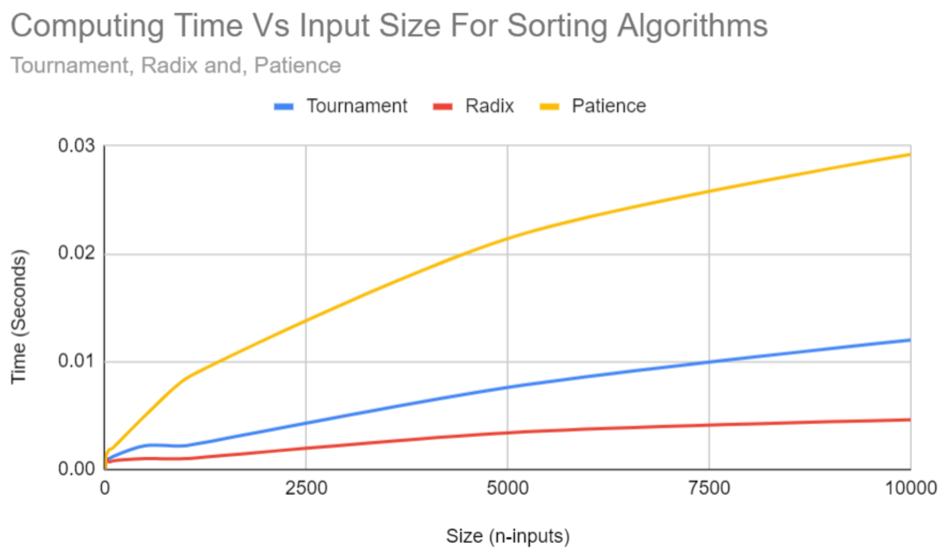


Figure 5: Shows a line for each algorithm which represents the average time it takes to sort each input size. Compares the Tournament Sort, Radix Sort and Patience Sort.

The final graph displays only the Tournament Sort, Radix Sort and Patience Sort because all have similar times to complete each input size that was tested. Of the 3, the Patience Sort took the longest and the Radix Sort was the quickest. Both the Patience Sort and Tournament Sort used a tree mechanism in order to find the next value to add to a sorted list while the Radix Sort preformed a series of mini sorts on each digit of the number.

In conclusion, this test provided great information about sorting techniques that can help others choose a sorting algorithm that will be efficient in sorting large amounts of data. The sorting algorithms that appear to be good candidates to achieve this based on the time it took to sort are the Radix Sort, Patience Sort, and the Tournament Sort because they had considerably lower run times compared to the rest of the other algorithms. The sorting algorithms that appear to be bad candidates to achieve this are the Burnt Pancake Sort, Gnome Sort, and the X+Y sort. It would be wise to avoid the Burnt Pancake Sort because it makes many unnecessary swaps that don't improve the running time of the sort. It would be wise to avoid the Gnome Sort because it lacks any recursion or sub-problems to limit the number of comparisons. Finally, avoid the X+Y sort due to thew large amount of values that have to be calculated from tall of the additions between the lists that are made which not only increases the amount of time that the sort takes but it can cause errors because it requires to much space to perform the sort.

Overall, these test runs provided great insight into why sorting is such an important topic in Computer Science and Software Engineering. As the amount of data that exists in the world continues grow as is created, efficient algorithms are need to ensure that the data is able to be sorted so that other calculations can be performed on it. This is why Mark Zuckerberg and other other prominent figures in computing place such an emphasis on the efficiency of their algorithms. This is also why it is important for people studying and entering the fields of Computer Science and Software Engineering should be focused on learning and understanding these algorithms in order to help push forward the next era of innovations in computing.

## References

[1] David Aldous and Persi Diaconis. Longest increasing subsequences: from patience sorting to the baik-deift-johansson theorem. *Bulletin of the American Mathematical Society*, 36(4):413–432, 1999.

[2] Walters G. Austin. Everyday algorithms: Pancake sort, Mar 2015.

[3] Tomek Czajka. Faster than radix sort: Kirkpatrick-reisch sorting, Jun 2020.

[4] GeeksforGeeks. Radix sort, Apr 2021.

[5] Dick Grune. Gnome sort - the simplest sort algorithm, 2003.

[6] Thomas Guest. Patience sort and the longest increasing subsequence, 2009.

[7] Karmella A Haynes, Marian L Broderick, Adam D Brown, Trevor L Butner, James O Dickson, W Lance Harden, Lane H Heard, Eric L Jessen, Kelly J Malloy, Brad J Ogden, et al. Engineering bacteria to solve the burnt pancake problem. *Journal of biological engineering*, 2(1):1–12, 2008.

[8] Joe James. *Tournament Sort Algorithm - a Heapsort variant.* YouTube, Oct 2015.

[9] Felix Richter. *Facebook Keeps On Growing.* Feb 2021.

[10] Steven S Skiena. *The algorithm design manual.* Springer International Publishing, 2020.

[11] Mark Zuckerberg. Cs50 lecture by mark zuckerberg, 2005.