Team Bagel

John Hacker, Nafisa Mostofa, Warren Atchison

We had many ideas throughout the development of our group-created program. One early idea included forming balanced binary tree or heap and then devolving it into an array or linked list to utilize the best runtimes of each data structure. This idea faced problems in light of uncertainty over the implementation and usefulness of the coding. We also theorized a sorting algorithm that would be seriously inefficient on purpose named Taz Sort. The idea of this algorithm was to randomly pick two indices in an array of unsorted numbers, swap the numbers, check if the array became sorted, and indefinitely repeat until it becomes sorted. This proposal gained affinity in the group for its Looney Tunes-inspired name. However, a complete lack of efficiency and difficulty with deciphering its Big-O runtime halted any progression to actually writing the code for such a program. Next, our team tried drawing inspiration from our bagel-themed group name by creating some sort of data structure that would loop around onto itself and be used for the purpose of sorting numbers efficiently. Such a structure would be possible by making a typical linked list, but having what would normally be considered the head and tail of the list be linked together for an infinite amount of traversals to the next node of a current node. Hours of hypothesizing and light drafting of code drew us to the conclusion that such an endeavor was not likely to bear us any success during the time that we had for this project. We kept trying to have the bagel data structure sort numbers in a way that was not wasteful while also being able to freely move around the supposed circle of linked nodes, passing on from the “tail” of the list directly to the “head” by simply going to forward in the list. All of our ideas failed due to a large variety of problems: too similar to another sorting type, only acts as a usual linked list without utilizing the circular linked-ness, wastefully traversed the circle so much that its worst case scenario would be dreadfully slow.

Finally, our group had a breakthrough in finding thinking of a sorting algorithm that is not super fast but is also not total terribly slow. We named this new idea Bounce Sort. The Big-O runtime is n-squared in the worst case and works in similar fashion to selection sort; but, in practicality it but works faster than selection sort and other n-squared sorting algorithms.

Originally, the way Bounce Sort works is that it compares the first and last values of the array that needs to be sorted. If the value at the first index of the array is larger than the value at the last index, the algorithm will switch the values. If not, then the array will stay the same. Then, Bounce Sort will recursively call the function without the first value and check if the new value at index one is larger than the last value. After the two largest values of the array are left, then Bounce Sort will recursively add the values back to the array in the order they were taken away and compare them to the second largest value. Then, when all the values are added back into the array, Bounce Sort will sort the rest of the elements in the array, disregarding the last two values, which are the largest ones. This will continue until the array is fully sorted.

We attempted to make the sorting more efficient by having a Big-O runtime of n log n, by dividing the array in groups of 3 numbers because that is the minimum amount of values required for bounce sort to work. However, this method was too similar to Merge Sort and did not make the runtime any faster. For future considerations, we would attempt to decrease the runtime by finding the two largest values of the array faster or by taking more values out of the array when we are sorting through it.

~~[illegible]~~ ~~[illegible]~~ ~~bounce (arr+1, size-1)~~

```
if(f > L){
  swap;}
```

|2|3|1|6|5| call bounce

```
bounce(arr+1, sz-1);
if(f > 2ndL)
  swap;
```

⇓

compare 2 to 5, no swap, ~~result~~ chop off 1st ~~[illegible]~~

|2|3|1|6|5|,

⇓

|3|1|6|5|

compare 3 to 5, no swap, chop off 1st

⇓

|1|6|5|

compare 1 to 5, no swap, chop off 1st

⇓

|6|5|

compare 6 to 5, swap, n=2, so return

⇓

going back up!

⇓

|1|5|6|

compare 1 to 5, no swap, add

⇓

|3|1|5|6|

compare 3 to 5, no swap, add

⇓

|2|3|1|5|6|

compare 2 to 5, no swap bounce complete

|2|3|1|5|6| call bounce on

|2|3|1|

~~bounce (arr+1, size),~~

bounce (arr+1, size-1)