Sary Hammad 1192698

Project #4

Data Structures and Algorithms

1. Gnome Sort:

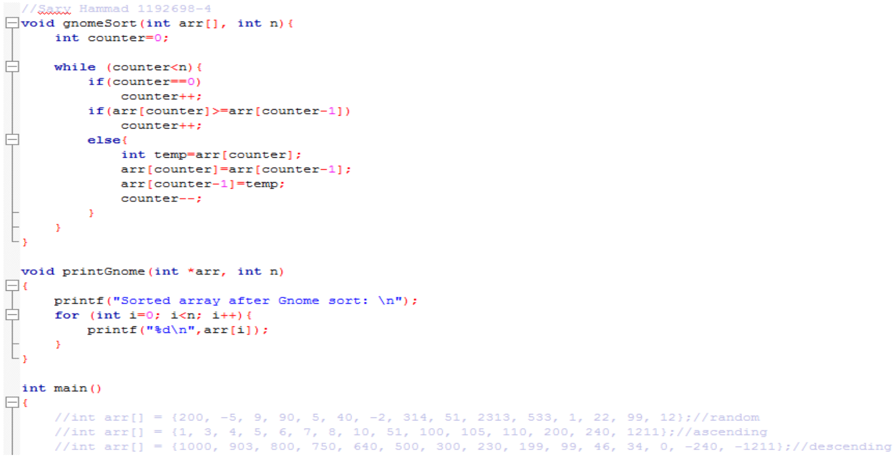
Gnome sort is a sorting algorithm nicknamed the stupid sort as it origins from how a farm gnome supposedly sorts his flowerpots. The algorithm goes like this:

While the counter does not equal the size of the array: (counter initially at 0)

- Checks if the counter is 0 (the first element of the array), and increments it
- Checks the arr[counter] with the element before it, and if they are in order it proceeds with incrementing the counter
- If the elements were not in order, the program swaps them and decrements the counter

When the counter is decremented, the loop repeats the condition checking, so if the order was disrupted with the swapping, it rechecks the element that is prior to the recently swapped term and places them back in order. The function keeps moving forward until the counter equates the size of the array, at which the entire array would have been sorted by then.

As the function uses a single loop, not a nested loop, it is expected to have a linear time complexity of $O(n)$, but it actually does not. The actual time complexity of this sorting algorithm is $O(n^2)$, and that is because the counter is not always incremented; it gets decremented too![1] The auxiliary space occupied by the algorithm is $O(1)$ (does not use extra storage), thus it is an in place algorithm.[2] The algorithm is also stable, as nothing but incrementing the counter is done when the two compared array items are equal. I have tested the algorithm on code blocks by inputting three different arrays of size 15.

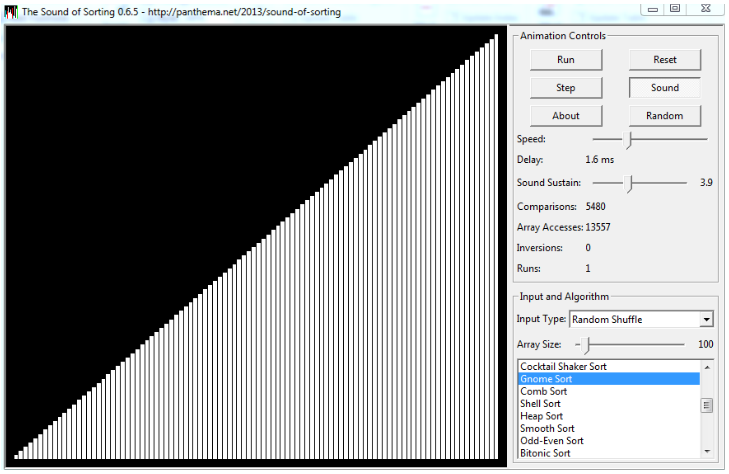


```
//Sary Hammad 1192698-4
void gnomeSort(int arr[], int n){
    int counter=0;

    while (counter<n){
        if(counter==0)
            counter++;
        if(arr[counter]>=arr[counter-1])
            counter++;
        else{
            int temp=arr[counter];
            arr[counter]=arr[counter-1];
            arr[counter-1]=temp;
            counter--;
        }
    }
}

void printGnome(int *arr, int n)
{
    printf("Sorted array after Gnome sort: \n");
    for (int i=0; i<n; i++){
        printf("%d\n",arr[i]);
    }
}

int main()
{
    //int arr[] = {200, -5, 9, 90, 5, 40, -2, 314, 51, 2313, 533, 1, 22, 99, 12};//random
    //int arr[] = {1, 3, 4, 5, 6, 7, 8, 10, 51, 100, 105, 110, 200, 240, 1211};//ascending
    //int arr[] = {1000, 903, 800, 750, 640, 500, 300, 230, 199, 99, 46, 34, 0, -240, -1211};//descending
```

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

[2] In-place vs out-of-place algorithms. Techie Delight. (2021, April 30). https://www.techiedelight.com/in-place-vs-out-of-place-algorithms/.

The algorithm does the average when the elements of the array are randomized (0.016 seconds). It supposedly does its best when the elements are sorted in ascending order already (but it took the same 0.016 seconds). And supposedly does its worse when the elements are in descending order (it took 0.044 seconds). The results were not consistent, therefore, I decided to use "Sounds of Sorting"[3], a simulator that visualizes different sorting algorithms and gives the number of array accesses and comparisons the algorithm does while sorting an array of 100 elements.



The Sound of Sorting 0.6.5 - http://panthema.net/2013/sound-of-sorting

Animation Controls

Speed:

Delay: 1.6 ms

Sound Sustain: 3.9

Comparisons: 5480

Array Accesses: 13557

Inversions: 0

Runs: 1

Input and Algorithm

Input Type: Random Shuffle

Array Size: 100

The results are: 13557AA+5480C (random/not sorted array), 100AA+99C (ascending array), 24751AA+9900C (descending).

2. Cocktail Sort:

Also called the cocktail shake sort, it is a sorting algorithm similar to the bubble sort in a sense that it places the largest numbers at last, but different from the bubble sort as it bidirectional, meaning that it also simultaneously places the smallest numbers first. Its algorithm goes like:[4]

- Loops through the array from left to right (forward pass)
- Compares each element with its next element, if array[i] is bigger than array[i+1], then they are swapped (biggest element is stored at the end)
- Loops through the array from right to left, starting from the last element in the forward pass (backward pass)
- Compares each element with the element before it, if array[i] is smaller than array[i-1], they are swapped (smallest element is stores at the start)
- These two iterations are repeated until the entire array is sorted, note that the forward pass also picks up from where the backward pass ended (at the smallest number).

[3] Bingmann, T. (2013, May 22). The Sound of Sorting - "Audibilization" and Visualization of Sorting Algorithms. panthemanet Weblog. https://panthema.net/2013/sound-of-sorting/.

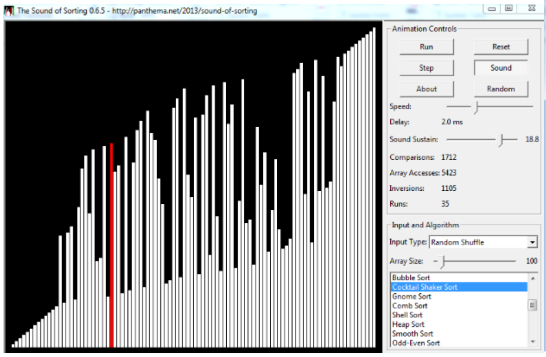
[4] KOMALASARI, C. (2021, April 19). A Comparative Study of Cocktail Sort and Insertion Sort. JACSM. https://jacsm.ro/view/?pid=31_3.

- The flag that ends this algorithm and signifies that the entire array is sorted is when there are no swaps done for a done loop, this is set as true at the start of the code, set as false when the loop starts, and is set back true when a swap is done.[5]

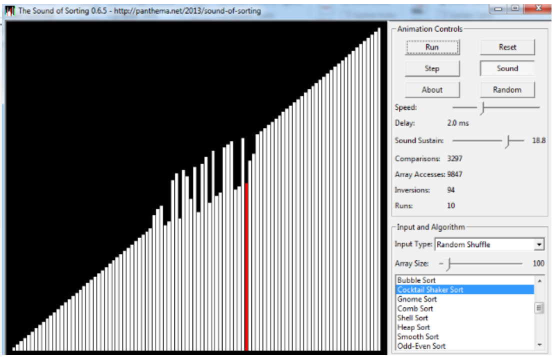
The cocktail sort uses two bubble sorts in two different iterations, thus it is more efficient than the bubble sort itself; it minimizes the number of elements it has to pass through by a factor of 2. The time complexity of the algorithm is $O(n^2)$, that is because it has two loops, one for each iteration, nested in a bigger while loop that checks if the swapping operation stopped so it can break the loop. The auxiliary space occupied by the algorithm is $O(1)$, thus it is an in place algorithm. It is also stable as two equal terms are never swapped in both of the iterations done. I used the same program to experiment on the algorithm and found out the following:

10057AA+3404R (random array), 298AA+198R (ascending), 17301AA+4950R (descending)

I also found out why it is called a cocktail shake sort, the way the elements are moved is like the algorithm is shaking the biggest to the right and the smallest to the left.


The Sound of Sorting 0.6.5 - http://panthema.net/2013/sound-of-sorting

Delay: 2.0 ms
Sound Sustain: 18.8
Comparisons: 1712
Array Accesses: 5423
Inversions: 1105
Runs: 35
Input Type: Random Shuffle
Array Size: 100
Cocktail Shaker Sort


The Sound of Sorting 0.6.5 - http://panthema.net/2013/sound-of-sorting

Delay: 2.0 ms
Sound Sustain: 18.8
Comparisons: 3297
Array Accesses: 9847
Inversions: 94
Runs: 10
Input Type: Random Shuffle
Array Size: 100
Cocktail Shaker Sort

3. Tim Sort:

This sorting algorithm is a very popular one since it was used as default in Java’s sorting methods (Arrays.sort()), as well as the default sorting method in Python. It is one of the quickest sorting algorithms out there, and even competes with the Quick sort in some cases. It is a combination of insert sort and the merge sort. The algorithm goes as follows:

- The array is divided into smaller arrays in which there is a default number of elements in each sub array 32 or sometimes 64, depending on the size of the array. (the array size is checked before it is split into different subarrays, if the size was smaller than 32, only insertion sort is used to sort the array).
- If the size is larger than 32, the algorithm proceeds.
- The program, with the help of a nested loop, loops through the elements of the sub-array, starting from the second element, comparing each element to the one before it.

[5] Sahai, H. (2019, June 10). Cocktail Shaker Sort / Bidirectional bubble sort. OpenGenus IQ: Learn Computer Science. https://iq.opengenus.org/cocktail-shaker-sort/.

- If the current element is smaller than the element before it, they are swapped, and the smaller number is compared to all the previous elements until the first element of the sub-array.
- After each sub array is sorted, the sub arrays are combined by the use of the merge method from the merge sort.
- Subarrays of size 32 are merged to form 64, then 128,etc..
- The entire array is sorted.

The time complexity of Tim sort is $O(nlog(n))$, the auxiliary space occupied by the algorithm is $O(n)$, that is because it uses a merge sort, and that algorithm uses extra space. It is stable however, as when sorting by the use of the two methods equal elements are in the same positions (both algorithms are stable). The results of simulating this algorithm go as follows:[6]

1714AA+550R (random), 100AA+99R (ascending), 199AA+99 (descending)

This algorithm realizes when a sub array is in reverse order and reverses it.

Comparing the algorithms:

We can hypothesize that the time needed for a comparison is the same as the time needed for array accesses, and it is a constant unit. If we took the sum of operations done in each randomized array, the gnome sort is worst, and then comes the cocktail sort, and then the tim sort. Comparing them in sorting ascending arrays, we find that the gnome sort matches the tim sort, while the cocktail sort lags behind, as it has to recheck the array after completion. Finally, comparing the runtimes of the descending arrays, which we expect to have been the worst scenario between the other two. We notice that the “stupid” gnome sort lagged behind the most, then comes the cocktail, and then the tim sort with a dominating short amount of time.

To conclude, the results were pretty much as expected, the tim sort excelled with its’ brilliant use of two algorithms smartly. The cocktail did not do that bad, it was also a great mixture of algorithms. The gnome really lived up to its name, being the stupid algorithm that has to always check back and forth, and has a time complexity of n squared with absolutely no advantage, there are still worse algorithms to be honest though! (the bogo sort is a mess!)

It is also important to mention that the expected time complexities were pretty accurate as well, as 100^2 is about 10000, which was the magnitude of the gnome and cocktail sorts in terms of the constant unit. 100log100 is about 1000, so that matches as well.

Note: the program used is extremely vivid (sounds and animations) that represent the sorting algorithms. It is best to watch a video such as this one in order to understand it (screenshots did not fully expose how nice of a program it is.

Cocktail sort visualized: https://www.youtube.com/watch?v=njClLBoEbfI

[6] TimSort. GeeksforGeeks. (2021, March 18). https://www.geeksforgeeks.org/timsort/.

References:

1. Belwariar, R. (2021, March 31). Gnome Sort. GeeksforGeeks. https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/
2. In-place vs out-of-place algorithms. Techie Delight. (2021, April 30). https://www.techiedelight.com/in-place-vs-out-of-place-algorithms/
3. Bingmann, T. (2013, May 22). The Sound of Sorting - "Audibilization" and Visualization of Sorting Algorithms. panthemanet Weblog. https://panthema.net/2013/sound-of-sorting/
4. KOMALASARI, C. (2021, April 19). A Comparative Study of Cocktail Sort and Insertion Sort. JACSM. https://jacsm.ro/view/?pid=31_3
5. Sahai, H. (2019, June 10). Cocktail Shaker Sort / Bidirectional bubble sort. OpenGenus IQ: Learn Computer Science. https://iq.opengenus.org/cocktail-shaker-sort/
6. TimSort. GeeksforGeeks. (2021, March 18). https://www.geeksforgeeks.org/timsort/
7. Merge Sort. GeeksforGeeks. (2021, May 18). https://www.geeksforgeeks.org/merge-sort/.