Department of Computer Science

COMP2421

Project 4: Sorting algorithms

~~~~~~

Student's Name: **Tariq Odeh**

Student's No: **1190699**

Sec: **4**

Instructor: Dr. Radi Jarrar

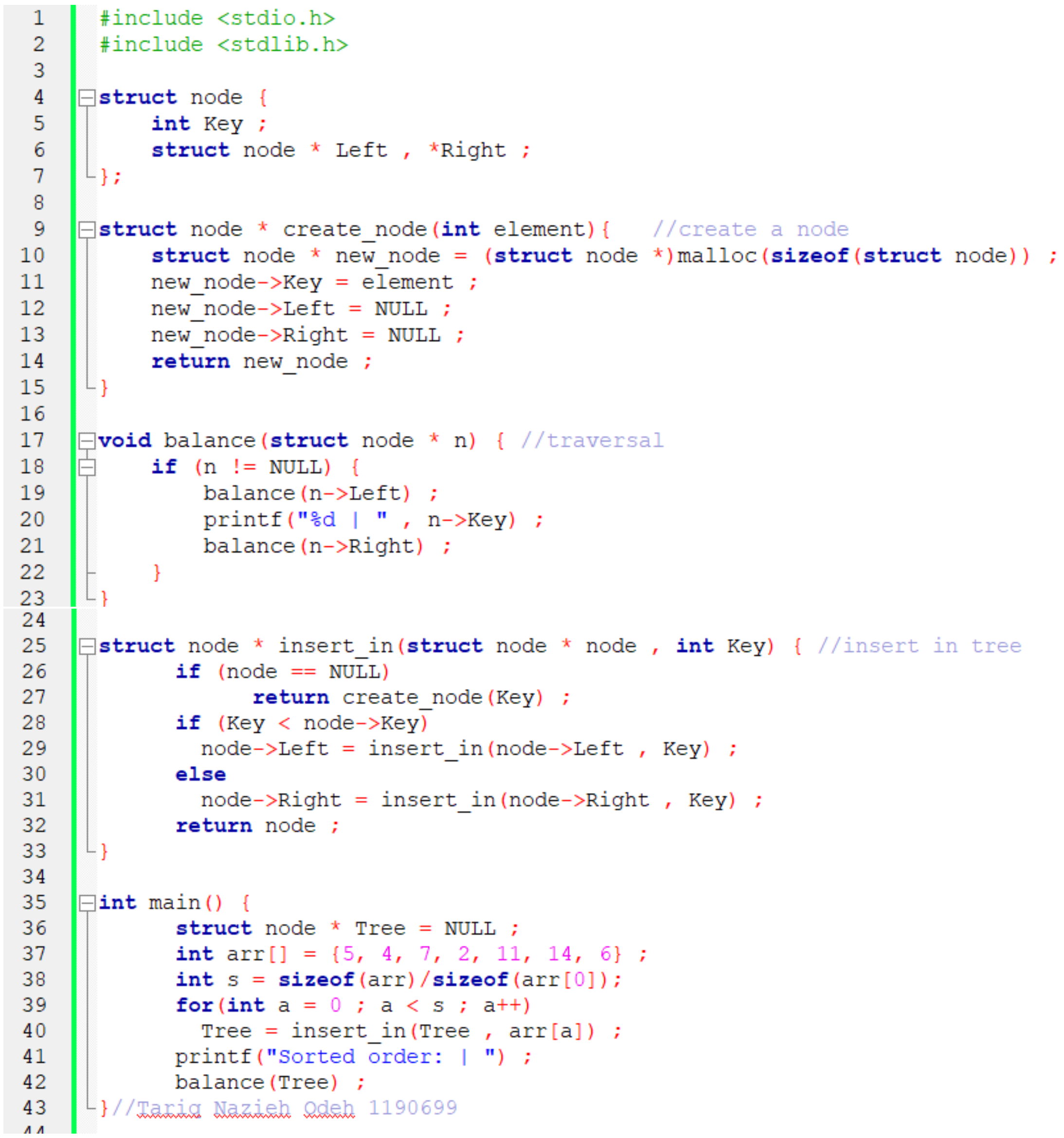
Date: 22-5-2021

~~~~~~

Tree Sort

Tree sort is a sorting algorithm that is rely on Binary Search Tree data structure. To perform the sort, it creates binary search tree from the element of array or input. whereas in binary search tree for every node N in the tree, the values of all the keys in its left subtree are smaller than the key value in N. And the values of all they key in its right subtree are larger than the key value in N. This means that all elements in the tree can be ordered in some consistent manner. Then the elements are extracted from Binary Search Tree depending on the order in which they were entered into, as for each node the values to the left of it are smaller than their value and the values on the right are greater than their value, and thus the elements is extracted in an ordered order.

Code:



```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int Key ;
    struct node * Left , *Right ;
};

struct node * create_node(int element){   //create a node
    struct node * new_node = (struct node *)malloc(sizeof(struct node)) ;
    new_node->Key = element ;
    new_node->Left = NULL ;
    new_node->Right = NULL ;
    return new_node ;
}

void balance(struct node * n) { //traversal
    if (n != NULL) {
        balance(n->Left) ;
        printf("%d | " , n->Key) ;
        balance(n->Right) ;
    }
}

struct node * insert_in(struct node * node , int Key) { //insert in tree
      if (node == NULL)
            return create_node(Key) ;
      if (Key < node->Key)
        node->Left = insert_in(node->Left , Key) ;
      else
        node->Right = insert_in(node->Right , Key) ;
      return node ;
}

int main() {
      struct node * Tree = NULL ;
      int arr[] = {5, 4, 7, 2, 11, 14, 6} ;
      int s = sizeof(arr)/sizeof(arr[0]);
      for(int a = 0 ; a < s ; a++)
        Tree = insert_in(Tree , arr[a]) ;
      printf("Sorted order: | ") ;
      balance(Tree) ;
}//Tariq Nazieh Odeh 1190699
```

Figure (1): C program to implement Tree Sort

Algorithm:

Step 1: Take the elements input in an array.

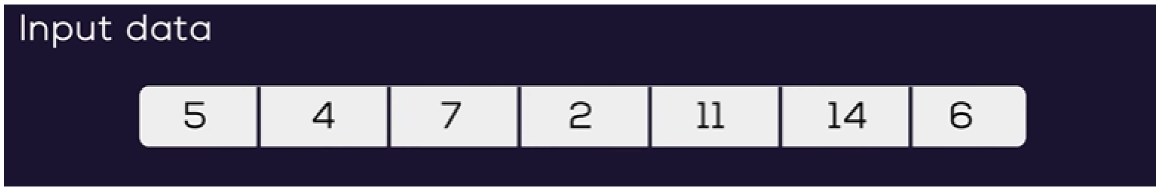


Figure (2): Input data

Step 2: Build the Binary search Tree by inserting elements from the array in Binary Search Tree.

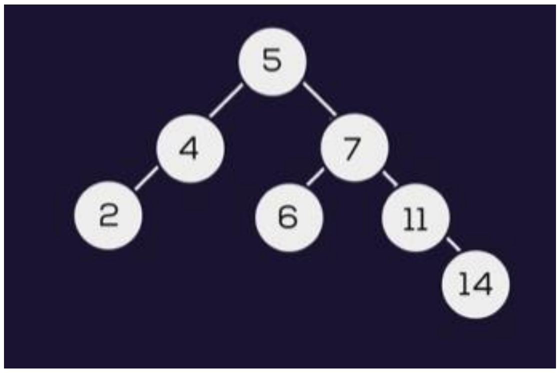


Figure (3): Data it in binary search tree

Step 3: Perform in-order traversal on the tree to get the elements back in sorted order.

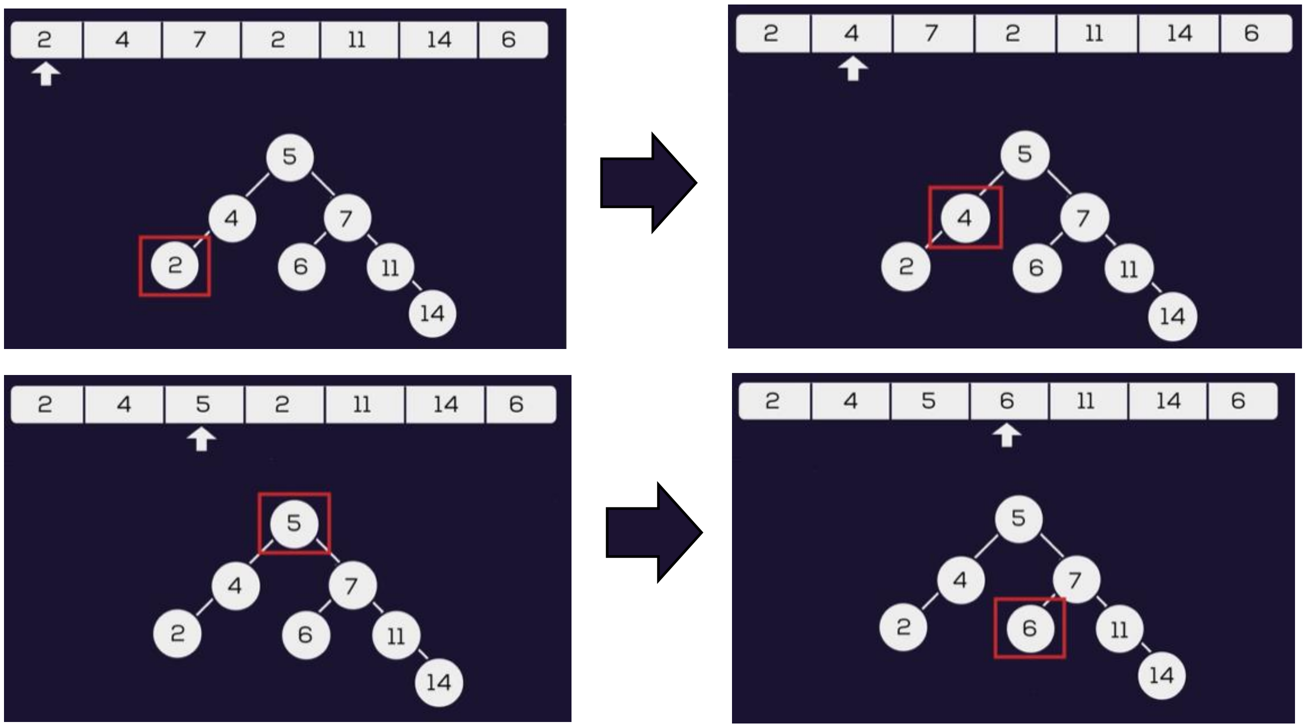


Figure (4): Traversal on the tree

And so on to get the sorted array:

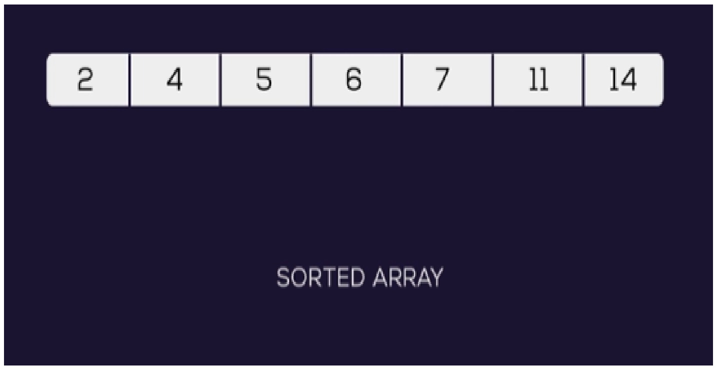


Figure (5): stored array

Properties:

- **Time complexity:** Average Case → O(n log n), Best Case → O(n log n): Adding one item to a Binary Search tree on average takes O(log n) time, adding n items will take O(n log n) time. Worst Case → O(n^2): When the algorithm operates on an already sorted**,** to insert 1 item, it needs to be compared to n items**,** to insert n items, we have to make n*n =n^2 comparisons.

- **Space Complexity:** O(n), because tree use to store data need memory for all tree nodes.

- **Stability:** Stable because the elements with identical values appear in the same order in the output array as they were in the input array.

- Adding n items is an O(n log n) process, making tree sorting a fast sort process, and we can make changes very easily as in a linked list.
  The worst case occur when the tree resembles a linked list, this worst case occurs when the algorithm operates on an already sorted, we can be improved by using a self-balancing binary search tree like AVL Tree. Using self-balancing binary tree Sort will take O(n log n) time to sort the array in worst case.

- Tree sort algorithms require separate memory to be allocated for the tree (memory for all tree nodes).

Running time of the algorithms if the input data array is:

- Sorted (ascending): To insert 1 item, it needs to be compared to n items

  To insert n items, we have to make n*n =n2 comparisons →O(n^2)

- Sorted (descending): To insert 1 item, it needs to be compared to n items

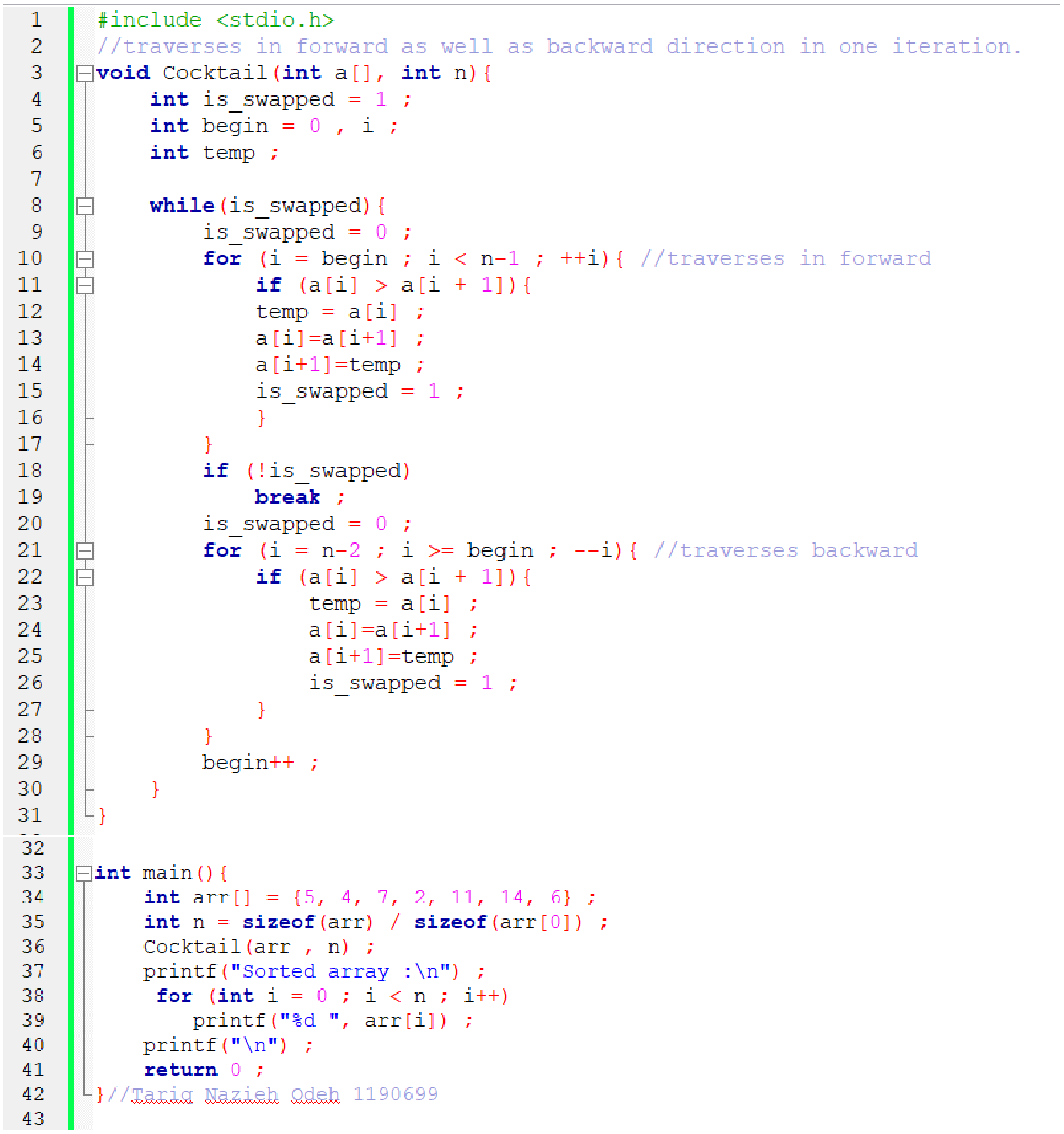
  To insert n items, we have to make n*n =n2 comparisons→O(n^2)

- Not sorted: To insert 1 item takes O(log n) time, adding n items will take O(n log n) time.

Cocktail Sort

Cocktail sort is the variation of bubble sort, that is both a stable sorting algorithm and a comparison sort, which traverses the list in both directions. Which is different from the Bubble sort which traverses the list in forward direction only, while this algorithm traverses in forward as well as backward direction in one iteration. This sorting algorithm (Cocktail sort) is only marginally more difficult to implement than a bubble sort.

Code:



```c
#include <stdio.h>
//traverses in forward as well as backward direction in one iteration.
void Cocktail(int a[], int n){
    int is_swapped = 1 ;
    int begin = 0 , i ;
    int temp ;

    while(is_swapped){
        is_swapped = 0 ;
        for (i = begin ; i < n-1 ; ++i){ //traverses in forward
            if (a[i] > a[i + 1]){
            temp = a[i] ;
            a[i]=a[i+1] ;
            a[i+1]=temp ;
            is_swapped = 1 ;
            }
        }
        if (!is_swapped)
            break ;
        is_swapped = 0 ;
        for (i = n-2 ; i >= begin ; --i){ //traverses backward
            if (a[i] > a[i + 1]){
                temp = a[i] ;
                a[i]=a[i+1] ;
                a[i+1]=temp ;
                is_swapped = 1 ;
            }
        }
        begin++ ;
    }
}

int main(){
    int arr[] = {5, 4, 7, 2, 11, 14, 6} ;
    int n = sizeof(arr) / sizeof(arr[0]) ;
    Cocktail(arr , n) ;
    printf("Sorted array :\n") ;
     for (int i = 0 ; i < n ; i++)
        printf("%d ", arr[i]) ;
    printf("\n") ;
    return 0 ;
}//Tariq Nazieh Odeh 1190699

```

Figure (6): C program to implement Cocktail sort

Algorithm:

Each iteration contains two parts traverses in forward and traverses backward:
The first part: loops like the Bubble Sort through the array from left to right. Throughout the loop the adjacent elements are compared, so that if the value on the left is greater the two elements are swapped, and at the end of the iteration the largest number of elements will be present at the end of the array.

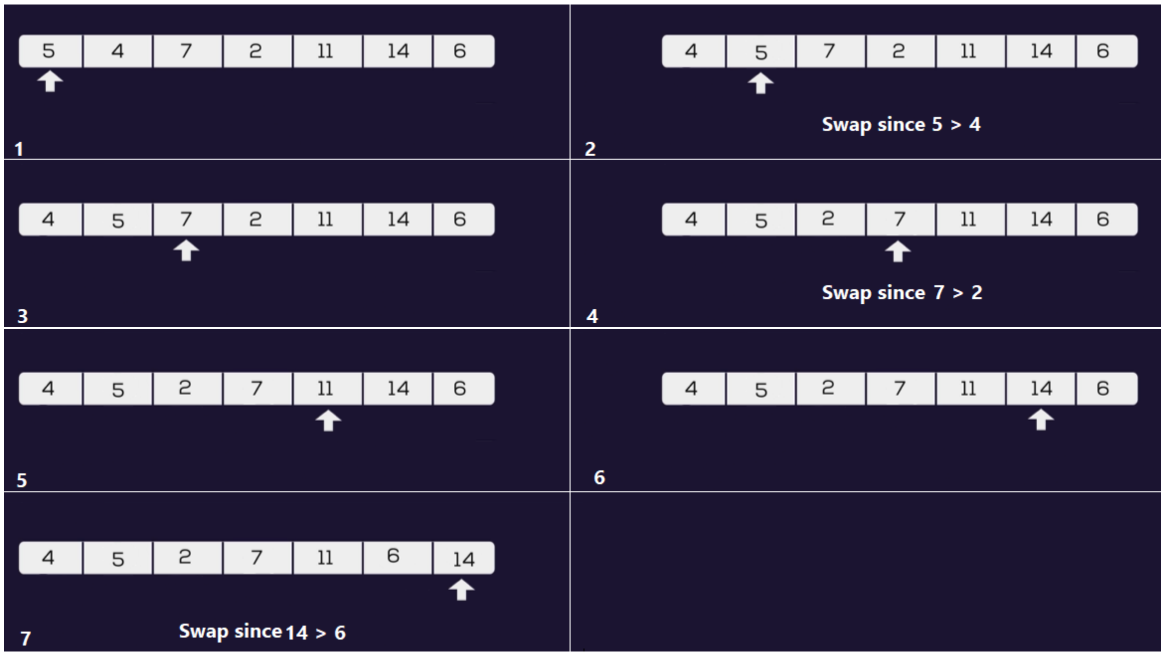


Figure (7): Traverses in forward

The second part: As in the first part, the array is compared in the opposite direction, starting from the element before the most recent element being sorted, until it reaches the beginning of the array, where the adjacent elements are compared, and here if the element on the right is smaller the two elements are swapped, and at the end of the iteration the smallest number of elements will be present the beginning of the array.

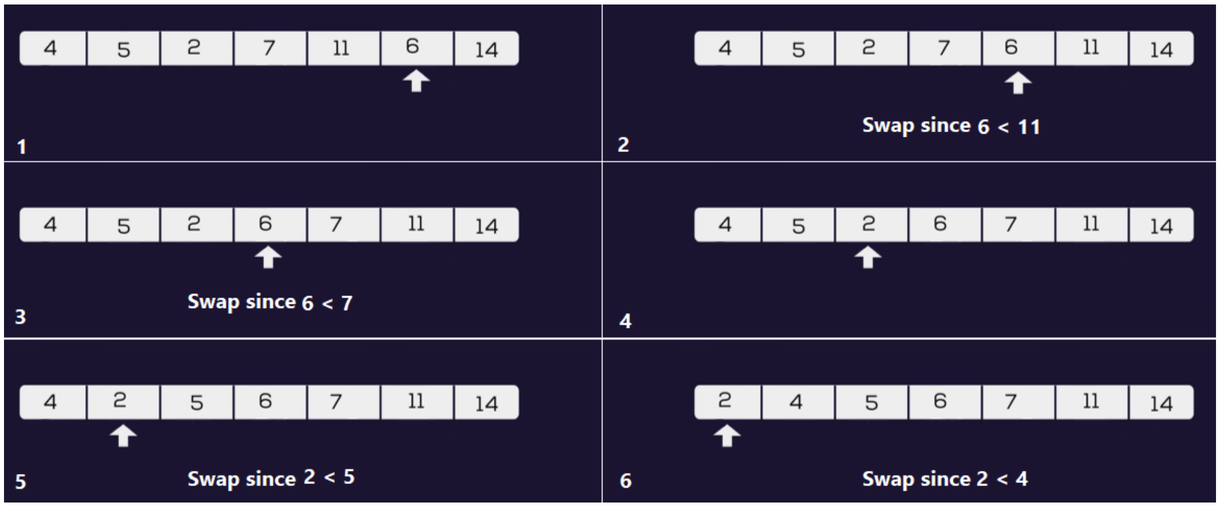


Figure (8): Traverses in backward

These traversed in both directions continue until we get the array sorted.

Note: Shaker sort like cocktail sort orders the array in both directions. But in the first one the smallest element ascends to the end of the array and in the second phase the largest element descends to the beginning of the array. Time complexity of Shaker sort like cocktail sort, but it differs in the case of sorting, where the best case occurs when the array is actually sorted (descending).

Properties:

- **Time complexity:** <u>Average Case</u> → O(n^2). <u>Worst Case</u> → O(n^2). Moves through the array and then back made n*n comparisons (each element in array is looked at twice).
  <u>Best Case</u> → O(n), best case occurs when array is already sorted (ascending). When complete first stage loops with no swap that mean the array is sorted and don't need another second stage loops (made n comparisons).

- **Space Complexity:** O(1), because an extra variable is used for swapping.

- **Stability:** Stable.

- Various efforts have been made to eliminate turtles to improve upon the speed of bubble sort. Cocktail sort is a bi-directional bubble sort that goes from beginning to end, and then reverses itself, going end to beginning. It can move turtles fairly well, but it retains O(n2) worst-case complexity.
  It used if complexity does not matter and short and simple code is preferred.

- No additional memory is required (Sorting in Place)

Running time of the algorithms if the input data array is:

- Sorted (ascending): In cocktail sort O(n) because when complete first stage loops with no swap that mean the array is sorted and don't need another second stage loops (made n comparisons). But in Shaker sort O(n^2) because each element in array is looked at twice.

- Sorted (descending: In cocktail sort O(n^2) because moves through the list and then back made n*n comparisons (each element in array is looked at twice). But in Shaker sort O(n) because when complete first stage loops with no swap that mean the array is sorted and don't need another second stage loops (made n comparisons).

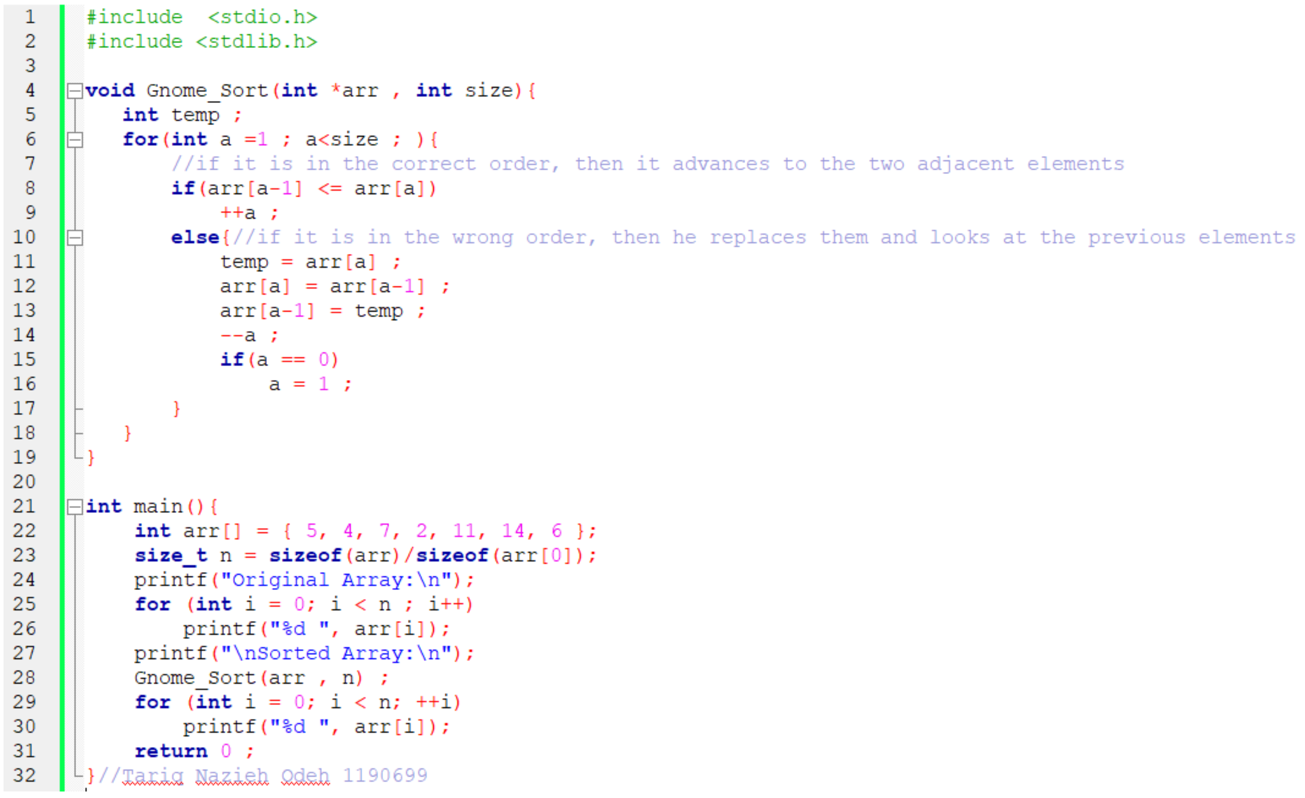
- Not sorted: O(n^2) → Moves through the list and then back made n*n comparisons (each element in array is looked at twice).

Gnome Sort

Gnome Sort also called Stupid sort It is conceptually simple, requiring no nested loops, where the key idea is to swap adjacent elements (if not in order) to sort the entire list. It is similar to Insertion sort except the fact that in this case, we swap adjacent elements.

It is inspired by the standard Dutch Garden Gnome sorting his flower pots. A garden gnome sorts the flower pots by the following method: Looking at the previous and adjacent flower pot, so that if it is in the correct order, then it advances to the two adjacent pots, but if it is in the wrong order, then he replaces them and looks at the previous pot (steps one pot backwards). He is at the starting of the pot line if there is no previous pot so he steps forwards, he is at the end of the pot line if here is no pot next to him so he is done.

Code:



```c
#include  <stdio.h>
#include <stdlib.h>

void Gnome_Sort(int *arr , int size){
    int temp ;
    for(int a =1 ; a<size ; ){
        //if it is in the correct order, then it advances to the two adjacent elements
        if(arr[a-1] <= arr[a])
            ++a ;
        else{//if it is in the wrong order, then he replaces them and looks at the previous elements
            temp = arr[a]  ;
            arr[a] = arr[a-1]  ;
            arr[a-1] = temp ;
            --a ;
            if(a == 0)
                a = 1 ;
        }
    }
}

int main(){
     int arr[] = { 5, 4, 7, 2, 11, 14, 6 };
     size_t n = sizeof(arr)/sizeof(arr[0]);
     printf("Original Array:\n");
     for (int i = 0; i < n ; i++)
         printf("%d ", arr[i]);
     printf("\nSorted Array:\n");
     Gnome_Sort(arr , n) ;
     for (int i = 0; i < n; ++i)
         printf("%d ", arr[i]);
     return 0 ;
}//Tariq Nazieh Odeh 1190699
```

Figure (9): C program to implement Gnome Sort

Algorithm:

Step 1: Go to the right element if you're at the beginning of the array.
Step 2: Go one step right if the current element is larger or equal to the previous element.
Step 3: Swap these two elements and go one step backwards if the current element is smaller than the prior element.
Step 4: Repeat steps 2 and 3 until the array reaches the end.
Step 5: If the end of the array is reached then stop and the array is sorted.

```
Array arr[]
Set elements to n
Set i to 1
While i less than or equal n
        if (arr[i] >= arr[i-1])
              i++;
       if (arr[i] < arr[i-1]){
             swap(arr[i], arr[i-1]);
             i--;  }
ENDWhile
Print arr
```

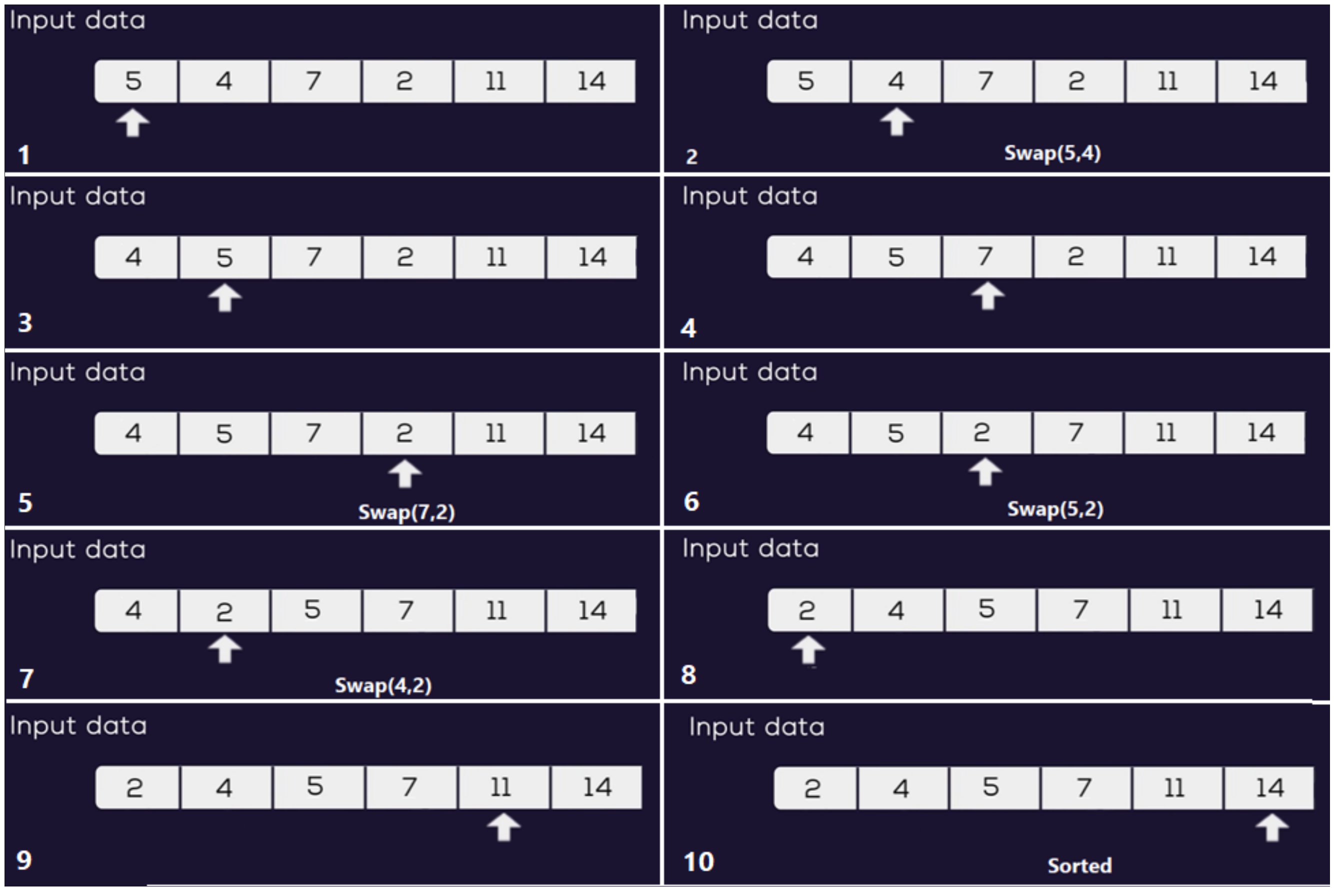


Figure (10): Example of Gnome Sort

Properties:

- **Time complexity:** Average Case → $O(n^2)$. Worst Case → $O(n^2)$. Because we don't always go to the next element, so the "index" in our program it gets decremented too, that made n*n comparisons.
  Best Case → O(n). If the matrix is sorted, we always go to the next element, so the "index" in our program gets increased, that made n comparisons.

- **Space Complexity:** O(1), because an extra variable is used for swapping.

- **Stability:** Stable.

- Gnome Sort values lies in teaching.

- No additional memory is required (Sorting in Place)

Running time of the algorithms if the input data array is:

- Sorted (ascending): O(n) → we always go to the next element, so the "index" in our program gets increased, that made n comparisons.

- Sorted (descending): $O(n^2)$ → Because we don't always go to the next element, so the "index" in our program it gets decremented too, that made n*n comparisons.

- Not sorted: $O(n^2)$ → Because we don't always go to the next element, so the "index" in our program it gets decremented too, that made n*n comparisons.

Summary

		Cocktail Sort	Tree Sort	Gnome Sort	Functions
Time complexity	Best Case	O(n)	O(n log n)	O(n)	O(n): Linear O(n log n): n-log-n $O(n^2)$: Quadratic
	Worst Case	$O(n^2)$	$O(n^2)$	$O(n^2)$	
	Average Case	$O(n^2)$	O(n log n)	$O(n^2)$	
Space Complexity		O(1)	O(n)	O(1)	O(1): no extra space O(n): extra space
Stability		Stable	Stable	Stable	
In Place		In Place	Not in Place	In Place	
Running time of the algorithms	Sorted (ascending)	O(n)	$O(n^2)$	O(n)	
	Sorted (descending)	$O(n^2)$	$O(n^2)$	$O(n^2)$	
	Not sorted	$O(n^2)$	O(n log n)	$O(n^2)$	

We can deduce from the preceding discussion and the summary table that:

- When array sorted (descending) Tree Sort, Gnome sort and Cocktail sort have the same time complexity $O(n^2)$.

- In the running time of the algorithms we can see that Tree sort is the fastest when data in the array are not sorted and Gnome sort and Cocktail sort have the same time, because in Tree sort one element takes O(log n) time, n elements will take O(n log n) time, but in Cocktail sort each element in array is looked at twice n*n and in Gnome sort we don't always go to the next element, that made n*n comparisons.

- Tree sorting it has a feature in that adding n items is an O(n log n) procedure, making it a rapid sort process with the ability to make changes as quickly as a linked list, but in Gnome sort and Cocktail sort need to insert element in array and then check sorted.

- Gnome sort and Cocktail sort sorting is done in same array so they don't need additional memory is required, but the Tree sort sorting is done in binary search tree so it need additional memory.

References

- Dr. Radi Jarrar Slides to get information about Asymptotic Time Analysis

- Geeks for geeks to get information about algorithm sort (Tree, Cocktail, Gnome)
  (accessed on 21/5 at 2:30 PM).
  https://www.geeksforgeeks.org/

- Sanfoundry to get code of CockTail Sort.
  (accessed on 21/5 at 12:00 PM).
  https://www.sanfoundry.com/c-program-implement-cocktail-sort/

- Java to point to know the time complexity of Gnome Sort
  (accessed on 22/5 at 11:00 AM).
  https://www.javatpoint.com/

- wikipedia to get information about CockTail Sort.
  (accessed on 22/5 at 1:30 PM).
  https://en.wikipedia.org/wiki/Cocktail_shaker_sort

- 2braces to get information about Tree Sort.
  (accessed on 23/5 at 2:00 PM).
  https://www.2braces.com/data-structures/tree-sort

- Delft stack to get time complexity of insert in Tree Sort.
  (accessed on 24/5 at 9:00 PM)
  https://www.delftstack.com/tutorial/algorithm/tree-sort/

- YouTube channel 0612 TV w/ NERDfirst to understand examples
  https://www.youtube.com/watch?v=hhrvM7__7fQ&list=PLJse9iV6ReqhrZyq301AtJuC6butyhmLY

Codes:

Tree sort

```
#include <stdio.h>

#include <stdlib.h>


struct node {

   int Key ;

   struct node * Left , *Right ;

};


struct node * create_node(int element){   //create a node

   struct node * new_node = (struct node *)malloc(sizeof(struct node)) ;

   new_node->Key = element ;

   new_node->Left = NULL ;

   new_node->Right = NULL ;

   return new_node ;

}


void balance(struct node * n) { //traversal

   if (n != NULL) {

      balance(n->Left) ;

      printf("%d | " , n->Key) ;

      balance(n->Right) ;

   }

}


struct node * insert_in(struct node * node , int Key) { //insert in tree

     if (node == NULL)

          return create_node(Key) ;
```

```
    if (Key < node->Key)
      node->Left = insert_in(node->Left , Key) ;
    else
      node->Right = insert_in(node->Right , Key) ;
    return node ;
}

int main() {
    struct node * Tree = NULL ;
    int arr[] = {5, 4, 7, 2, 11, 14, 6} ;
    int s = sizeof(arr)/sizeof(arr[0]);
    for(int a = 0 ; a < s ; a++)
      Tree = insert_in(Tree , arr[a]) ;
    printf("Sorted order: | ") ;
    balance(Tree) ;
}//Tariq Nazieh Odeh 1190699
```

Cocktail sort

```
#include <stdio.h>

//traverses in forward as well as backward direction in one iteration.

void Cocktail(int a[], int n){

   int is_swapped = 1 ;

   int begin = 0 , i ;

   int temp ;


   while(is_swapped){

      is_swapped = 0 ;

      for (i = begin ; i < n-1 ; ++i){ //traverses in forward

         if (a[i] > a[i + 1]){

         temp = a[i] ;

         a[i]=a[i+1] ;

         a[i+1]=temp ;

         is_swapped = 1 ;

         }

      }

      if (!is_swapped)

         break ;

      is_swapped = 0 ;

      for (i = n-2 ; i >= begin ; --i){ //traverses backward

         if (a[i] > a[i + 1]){

            temp = a[i] ;

            a[i]=a[i+1] ;

            a[i+1]=temp ;

            is_swapped = 1 ;

         }
```

```
        }
        begin++ ;
    }
}

int main(){
    int arr[] = {5, 4, 7, 2, 11, 14, 6} ;
    int n = sizeof(arr) / sizeof(arr[0]) ;
    Cocktail(arr , n) ;
    printf("Sorted array :\n") ;
     for (int i = 0 ; i < n ; i++)
        printf("%d ", arr[i]) ;
    printf("\n") ;
    return 0 ;
}//Tariq Nazieh Odeh 1190699
```

Gnome sort

```
#include  <stdio.h>

#include <stdlib.h>

void Gnome_Sort(int *arr , int size){

  int temp ;

  for(int a =1 ; a<size ; ){

     //if it is in the correct order, then it advances to the two adjacent elements

     if(arr[a-1] <= arr[a])

        ++a ;

     else{//if it is in the wrong order, then he replaces them and looks at the
previous elements

        temp = arr[a] ;

        arr[a] = arr[a-1] ;

        arr[a-1] = temp ;

        --a ;

        if(a == 0)

           a = 1 ;

   }}}

int main(){

   int arr[] = { 5, 4, 7, 2, 11, 14, 6 };

   size_t n = sizeof(arr)/sizeof(arr[0]);

  printf("Original Array:\n");

  for (int i = 0; i < n ; i++)

     printf("%d ", arr[i]);

  printf("\nSorted Array:\n");

   Gnome_Sort(arr , n) ;

   for (int i = 0; i < n; ++i)

            printf("%d ", arr[i]);

   return 0 ;

}//Tariq Nazieh Odeh 1190699
```