Project Report

Introduction

Algorithms Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

For example, we might need to sort a sequence of numbers into nondecreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools.

Importance of Algorithms

1. To improve the efficiency of a computer program

   In programming, there are different ways of solving a problem. However, the efficiency of the methods available vary. Some methods are well suited to give more accurate answers than others. Algorithms are used to find the best possible way of solving a problem. In doing so they improve the efficiency of a program.

2. Proper utilization of resources

   A typical computer has different resources. One of them is computer memory. During the execution phase, a computer program will require some amount of memory. Some programs use more memory space than others. The usage of computer memory depends on the algorithm that has been used.

3. Algorithms

   It will be imperative to look at the cost. This is because each resource comes with a price tag. One can decide to use an algorithm that will use the least resources. The leaner the resources, the less the cost.

Motivation

Since algorithms are very important to increase one's efficiency, our project aims to make understanding them easy by visualizing the sorting and searching algorithms.

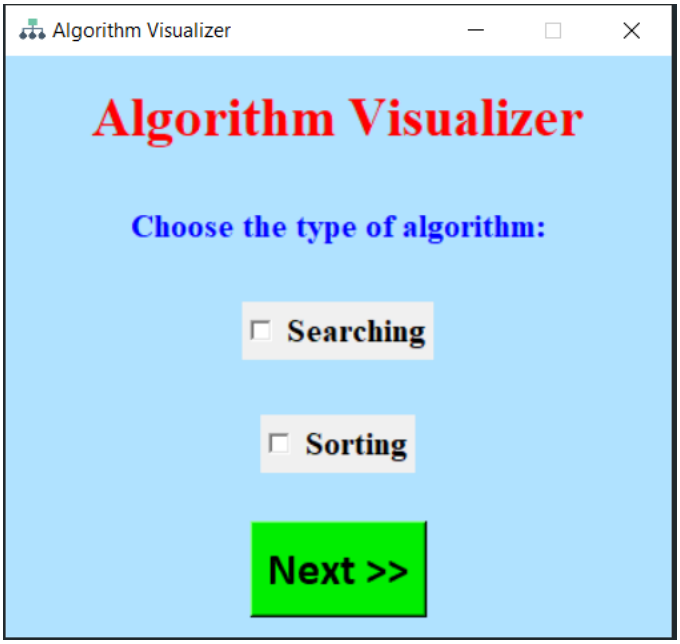
Algorithm Visualizers

A graphical user interface (**GUI**) to visualize different **searching (binary search, linear search)** and **sorting algorithms (merge sort, quick sort, insertion sort, bubble sort)** using python library **Tkinter**.

Implementation

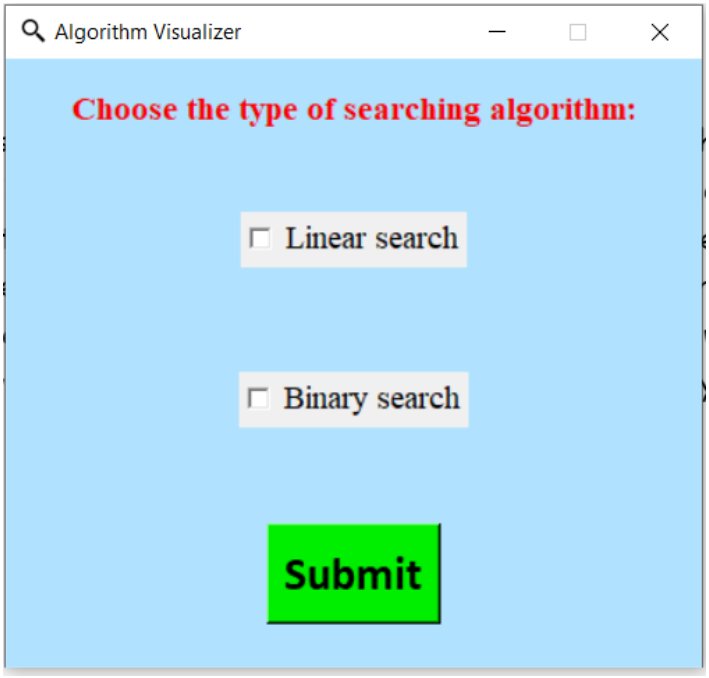
Main window

The first window is dedicated to select the algorithm type (searching or sorting). There are 2 checkboxes and each one corresponds to an algorithm type. The user is expected to select the type and click on the "Next" button. The "Next" button will take you to a new window depending on the algorithm type selected. If either more than one type is selected or if none is selected an error message pops up. For exit, it will show a warning message to check if you really want to exit or not. Figure below shows the screenshot of the main window.



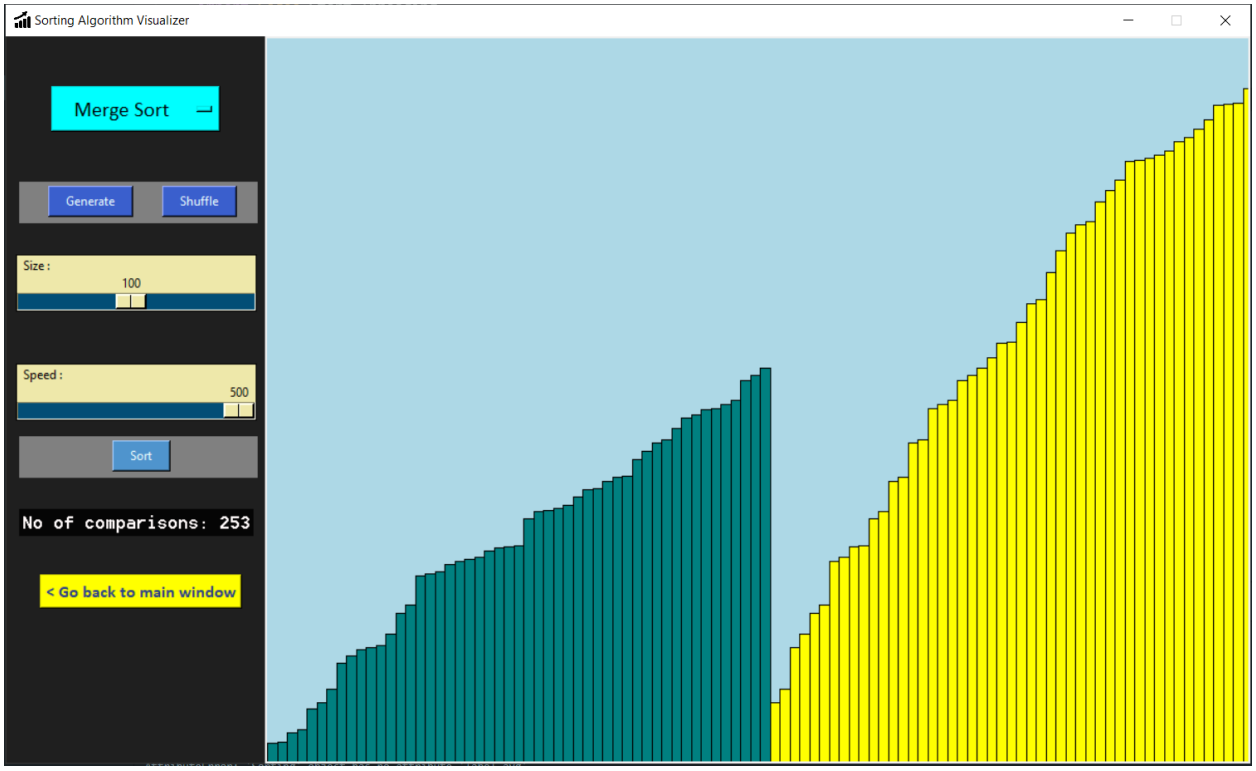
Second Window

This window contains checkboxes corresponding to the type of algorithm selected in the previous window. The user is expected to select any one from the list and click on the "Submit" button. The "Submit" button will take you to a new window depending on the algorithm selected. The error and the warning messages are similar to that of the previous window. Figure below shows the screenshot of this window when the "Searching" algorithm is chosen by the user.



Third Window

The third window is the visualizing window as per the choice of the user in the second window. It let's the user generate and shuffle an array to visualize the particular algorithm on it, the size of the array can be decided by the user by using the slider provided. The user can also change the speed of visualization of the algorithm. It also shows the number of comparisons throughout the execution of the particular algorithm so that one can understand why a particular algorithm is better than the other. This window also gives the user an option to go back to the main menu and choose another algorithm. Figure below is the snapshot of the Merge Sorting algorithm.



Searching Algorithms

Linear Search

A simple approach is to do **linear search**, i.e

1. Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
2. If x matches with an element, return the index.
3. If x doesn't match with any of the elements, print "Element not found".

The time complexity of the above algorithm is O(n).

```
# If you want to implement Linear Search in python

# Linearly search x in arr[]
# If x is present then return its location
```

```
# else return -1

def search(arr, x):

    for i in range(len(arr)):

        if arr[i] == x:
            return i

    return -1
```

Binary Search

This search algorithm takes advantage of a collection of elements that is already sorted by ignoring half of the elements after just one comparison.

1. Compare x with the middle element.
2. If x matches with the middle element, we return the mid index.
3. Else if x is greater than the mid element, then x can only lie in the right (greater) half subarray after the mid element. Then we apply the algorithm again for the right half.
4. Else if x is smaller, the target x must lie in the left (lower) half. So we apply the algorithm for the left half.

The time complexity of the above algorithm is O(log n).

```python
# Python 3 program for recursive binary search.
# Modifications needed for the older Python 2 are found in
comments.

# Returns index of x in arr if present, else -1
def binary_search(arr, low, high, x):

    # Check base case
    if high >= low:

        mid = (high + low) // 2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it can only
        # be present in left subarray
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)

        # Else the element can only be present in right subarray
        else:
            return binary_search(arr, mid + 1, high, x)

    else:
        # Element is not present in the array
        return -1

# Test array
arr = [ 2, 3, 4, 10, 40 ]
x = 10

# Function call
result = binary_search(arr, 0, len(arr)-1, x)
```

```
if result != -1:
    print("Element is present at index", str(result))
else:
    print("Element is not present in array")
```

Sorting Algorithms

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

Average Time complexity for this algorithm is $O(n^2)$.

```
# Python program for implementation of Bubble Sort

def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]
```

```
bubbleSort(arr)

print ("Sorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i])
```

Merge Sort

It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one. See the following C implementation for details.

Average Time complexity for this algorithm is O(n log(n)).

```
MergeSort(arr[], l,  r)
If r > l
    1. Find the middle point to divide the array into two halves:
            middle m = l+ (r-l)/2
    2. Call mergeSort for first half:
            Call mergeSort(arr, l, m)
    3. Call mergeSort for second half:
            Call mergeSort(arr, m+1, r)
    4. Merge the two halves sorted in step 2 and 3:
            Call merge(arr, l, m, r)
```

```
# Python program for implementation of MergeSort
def mergeSort(arr):
    if len(arr) > 1:

         # Finding the mid of the array
        mid = len(arr)//2
```

```
        # Dividing the array elements
        L = arr[:mid]

        # into 2 halves
        R = arr[mid:]

        # Sorting the first half
        mergeSort(L)

        # Sorting the second half
        mergeSort(R)

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

# Code to print the list
```

```
def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()


# Driver Code
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is", end="\n")
    printList(arr)
    mergeSort(arr)
    print("Sorted array is: ", end="\n")
    printList(arr)
```

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Average Time complexity for this algorithm is $O(n^2)$.

```
# Python program for implementation of Insertion Sort

# Function to do insertion sort
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1
        arr[j + 1] = key

# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
for i in range(len(arr)):
    print ("% d" % arr[i])
```

Quick Sort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick the first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.

4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

AverageTime complexity for this algorithm is O(n log(n)).

```
# Python3 implementation of QuickSort

# This Function handles sorting part of quick sort
# start and end points to first and last element of
# an array respectively
def partition(start, end, array):

    # Initializing pivot's index to start
    pivot_index = start
    pivot = array[pivot_index]

    # This loop runs till start pointer crosses
    # end pointer, and when it does we swap the
    # pivot with element on end pointer
    while start < end:

        # Increment the start pointer till it finds an
        # element greater than  pivot
        while start < len(array) and array[start] <= pivot:
            start += 1

        # Decrement the end pointer till it finds an
        # element less than pivot
        while array[end] > pivot:
            end -= 1
```

```
        # If start and end have not crossed each other,
        # swap the numbers on start and end
        if(start < end):
            array[start], array[end] = array[end], array[start]

    # Swap pivot element with element on end pointer.
    # This puts pivot on its correctly sorted place.
    array[end], array[pivot_index] = array[pivot_index],
array[end]

    # Returning end pointer to divide the array into 2
    return end

# The main function that implements QuickSort
def quick_sort(start, end, array):

    if (start < end):

        # p is partitioning index, array[p]
        # is at right place
        p = partition(start, end, array)

        # Sort elements before partition
        # and after partition
        quick_sort(start, p - 1, array)
        quick_sort(p + 1, end, array)

# Driver code
array = [ 10, 7, 8, 9, 1, 5 ]
quick_sort(0, len(array) - 1, array)

print(f'Sorted array: {array}')
```

Real-Life Application of Algorithms

1. The contact book in our phones **needs** to be **sorted** alphabetically so that we can find a person's phone number easily. Some other **examples are** flights, movie tickets; they **are** all **sorted** by **time**.
2. Imagine you are in a library, there are thousands of books to go through to find the exact book you are looking for. How can we sort to be able to search through to find the one we want? The Library came up with a sorting algorithm used to not only keep the books organized, but also to make it easier to find books.