Sorting Algorithms

Computational Thinking with Algorithms - G00241862 - Project 2019

What is a sorting Algorithm?

A sorting algorithm is an algorithm that takes elements of a list and arranges them in a certain order. Sorting is a fundemental aspect of coding as many more advanced algorithms require data to be sorted before they can function.

Specifically, sorting algorithms do not alter the elements in a list, it is only the order in which they are listed that changes and also, the order is non decreasing in that each element is no smaller than the previous element.

Sorting algorithms should be stable. This means that if two elements are correctly ordered in the list before the sort, then they must still be in the correct order subsequent to the sort.

Deciding what algorithm to choose depends on a number of factors as each one is suited to different types and sizes of data. If an inappropriate algorithm is chosen for a sorting task it may take a long time slowing down a program of which it forms a part. Factors that can influence the algorithm chosen include number of items to be sorted, extent towhich the items are already sorted, the level of stability required and how much code the user wants to write.

For the purpose of this exercise, we are going to use randomly generated integer arrays of different sizes and each sorting algorithm will put them in numerical order and will be tested for their respective time efficiency in performing the sort.

The sorting algorithms chosen to benchmark are Bubble Sort, Selection Sort, Insertion Sort, Quick Sort and Counting Sort.

The first three algorithms are simple comparison-based sorts, Quick Sort is an example of efficient compariso-based sort and , finally, Counting Sort, which is an example of non-comparison sort.

https://en.wikipedia.org/wiki/Sorting_algorithm (https://en.wikipedia.org/wiki/Sorting_algorithm)

https://medium.com/@george.seif94/a-tour-of-the-top-5-sorting-algorithms-with-python-code-43ea9aa02889 (https://medium.com/@george.seif94/a-tour-of-the-top-5-sorting-algorithms-with-python-code-43ea9aa02889)

https://www.geeksforgeeks.org/sorting-algorithms/ (https://www.geeksforgeeks.org/sorting-algorithms/)

In [86]:

```
import time # to allow for the timing of each algorithm to be recorded
import random # to import random data to sort
import numpy as np #to import numpy
from random import randint # to import random integers to be imported
import matplotlib.pyplot as plt # to allow for the visual plotting of the results
import seaborn as sns
import pandas as pd
```

Bubble Sort

Bubble Sort is very simple in process. It work by starting at the beginning of the list and comparing the first two elements. If the second element is smaller than the first, it swaps them so the smaller element is first. It then continues to compare the second and third. If the third element is smaller it swaps, if it is larger it does not swap. This continues until the end of the list is reached. At this point, the algorithm returns to the begining of the list and repeats the process. It keeps on repeating until it goes through the list without having to make any swaps.

Bubble Sort has best case efficiency of n, worst case efficiency of n squared and average case of n squared, a space complexity of 1 and is a Stable algorithm.

https://www.geeksforgeeks.org/bubble-sort/ (https://www.geeksforgeeks.org/bubble-sort/)

https://interactivepython.org/runestone/static/pythonds/SortSearch/TheBubbleSort.html (https://interactivepython.org/runestone/static/pythonds/SortSearch/TheBubbleSort.html)

https://www.youtube.com/watch?v=YHm_4bVOe1s (https://www.youtube.com/watch?v=YHm_4bVOe1s)

https://www.pythoncentral.io/bubble-sort-implementation-guide/ (https://www.pythoncentral.io/bubble-sort-implementation-guide/)

In [87]:

```
#Define the bubble sorting algorithm
bstart = time.time() # set the start time
n = 10 # the function is to run 10 times
def bubsort(x):
    for i in range(len(x)-1,0,-1):  #Setting the range for comparison
        for j in range(i):  #Comparing within set range
            if x[j]>x[j+1]:  # compare two adjacent elements
                temp = x[j]
                x[j] = x[j+1] # swap the elements
                x[j+1] = temp # put the smaller element in new position and loop around
again
for _ in range(n):
    array = np.random.randint(100, size=100)
    bubsort (array)
bend = time.time() # end the time
bstart1 = time.time()
for _ in range(n):
    array1 = np.random.randint(100, size=500)
    bubsort (array1)
bend1 = time.time()
bstart2 = time.time()
for _ in range(n):
    array2 = np.random.randint(100, size=1000)
    bubsort (array2)
bend2 = time.time()
bstart3 = time.time()
for _ in range(n):
    array3 = np.random.randint(100, size=2000)
    bubsort (array3)
bend3 = time.time()
bstart4 = time.time()
for _ in range(n):
    array4 = np.random.randint(100, size=5000)
    bubsort (array4)
bend4 = time.time()
bstart5 = time.time()
for _ in range(n):
    array5 = np.random.randint(100, size=10000)
    bubsort (array5)
bend5 = time.time()
```

Selection Sort

With Selection Sort, the Algorithm iterates through the list until it has established the smallest element. Once this is established, it moves that element into the first position in the list. This becomes the sorted part of the list. It thhen iterates through the list again to find the smallest value in the unsorted part of the list. Once this element is established it moves that element to the second position in the list. The sorted part of the list is now the first two elements. The algorithm continues to loop through the process until the sorted part of the list is the entire list.

Selection Sort has best case efficiency of n squared, worst case efficiency of n squared and average case of n squared, a space complexity of 1 and is not a Stable algorithm.

https://interactivepython.org/courselib/static/pythonds/SortSearch/TheSelectionSort.html (https://interactivepython.org/courselib/static/pythonds/SortSearch/TheSelectionSort.html)

https://www.geeksforgeeks.org/python-program-for-selection-sort/ (https://www.geeksforgeeks.org/python-program-for-selection-sort/)

https://www.youtube.com/watch?v=mI3KgJy_d7Y (https://www.youtube.com/watch?v=mI3KgJy_d7Y)

http://codecry.com/python/selection-sort (http://codecry.com/python/selection-sort)

In [88]:

```
#Define the selection sorting algorithm
sstart = time.time()
n = 10 # the function is to run 10 times
def selsort(x):
    for i in range (0, len(x)-1):
        minpos = i
        for j in range (i+1,len(x)):
            if x [j] < x [minpos]:
                minpos = j
        if minpos != i:
            x[i], x[minpos] = x[minpos], x[i]
for _ in range(n):
    array = np.random.randint(100, size=100)
    selsort (array)
send = time.time()
sstart1 = time.time()
for _ in range(n):
    array1 = np.random.randint(100, size=500)
    selsort (array1)
send1 = time.time()
sstart2 = time.time()
for _ in range(n):
    array2 = np.random.randint(100, size=1000)
    selsort (array2)
send2 = time.time()
sstart3 = time.time()
for _ in range(n):
    array3 = np.random.randint(100, size=2000)
    selsort (array3)
send3 = time.time()
sstart4 = time.time()
for _ in range(n):
    array4 = np.random.randint(100, size=5000)
    selsort (array4)
send4 = time.time()
sstart5 = time.time()
for _ in range(n):
    array5 = np.random.randint(100, size=10000)
    selsort (array5)
send5 = time.time()
```

Insertion Sort

With Insertion Sort, the algorithm goes through each element in a list, starting on the second element as there is nowhere left of the first element to compare to. Each time it comes to an element, if it is lesser than the previous, it takes the element and inserts it in the correct position behind it. By the time the algorithm reaches the end of the list, it inserts the final element in the correct position and the full list is now sorted.

Insertion Sort has best case efficiency of n, worst case efficiency of n squared and average case of n squared, a space complexity of 1 and is a Stable algorithm.

https://www.geeksforgeeks.org/insertion-sort/ (https://www.geeksforgeeks.org/insertion-sort/)

https://www.codesdope.com/blog/article/sorting-a-list-using-insertion-sort-in-python/ (https://www.codesdope.com/blog/article/sorting-a-list-using-insertion-sort-in-python/)

https://tutorialedge.net/compsci/sorting/insertion-sort-in-python/ (https://tutorialedge.net/compsci/sorting/insertion-sort-in-python/)

https://www.youtube.com/watch?v=Nkw6Jg_Gi4w (https://www.youtube.com/watch?v=Nkw6Jg_Gi4w)

In [89]:

```
#Define the insertion sorting algorithm
istart = time.time()
n = 10 # the function is to run 10 times
def inssort(x):
    for i in range (1, len(x)): # loop through the list starting at the second element.
        currvalue = x[i]
        temp = i
        while temp>0 and x[temp-1]>currvalue:
            x[temp] = x[temp-1]
            temp = temp-1

        x[temp] = currvalue
for _ in range(n):
    array = np.random.randint(100, size=100)
    inssort (array)
iend = time.time()
istart1 = time.time()
for _ in range(n):
    array1 = np.random.randint(100, size=500)
    inssort (array1)
iend1 = time.time()
istart2 = time.time()
for _ in range(n):
    array2 = np.random.randint(100, size=1000)
    inssort (array2)
iend2 = time.time()
istart3 = time.time()
for _ in range(n):
    array3 = np.random.randint(100, size=2000)
    inssort (array3)
iend3 = time.time()
istart4 = time.time()
for _ in range(n):
    array4 = np.random.randint(100, size=5000)
    inssort (array4)
iend4 = time.time()
istart5 = time.time()
for _ in range(n):
    array5 = np.random.randint(100, size=10000)
    inssort (array5)
iend5 = time.time()
```

Quick Sort

Quicksort is a recursive divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

First step is to establish a pivot, this can be the first element or the last element or, more effectively, by getting the median of three elements.

Then reorder the list so that elements less than the pivot come before the pivot and element greater than the array come after ithe pivot. This is called partitioning and leaves the pivot in the correct position.

Then recursively re run the process on both sides of the pivot until the full list is sorted.

Quick Sort has best case efficiency of n log n, worst case efficiency of n squared and average case of n log n, a space complexity of n and is not a Stable algorithm.

https://en.wikipedia.org/wiki/Quicksort (https://en.wikipedia.org/wiki/Quicksort)

https://www.geeksforgeeks.org/quick-sort/ (https://www.geeksforgeeks.org/quick-sort/)

https://www.youtube.com/watch?v=CB_NCoxzQnk (https://www.youtube.com/watch?v=CB_NCoxzQnk)

In [90]:

```
# Define the Quicksort Algorithm
qstart = time.time()
n = 10 # the function is to run 10 times
def quisort(x):
    quisort2(x, 0, len(x)-1)

def quisort2(x, low, hi):
    if low < hi:
        p = partition (x, low, hi)
        quisort2(x, low, p-1)
        quisort2(x, p+1, hi)
def get_pivot(x, low, hi):
    mid = (hi + low)//2
    pivot = hi
    if x[low] < x[mid]:
        if x[mid] < x[hi]:
            pivot = mid
    elif x[low] < x[hi]:
        pivot = low
    return pivot
def partition (x, low, hi):
    pivotIndex = get_pivot(x, low, hi)
    pivotValue = x[pivotIndex]
    x[pivotIndex], x[low] = x[low], x[pivotIndex]
    border = low

    for i in range(low, hi+1):
        if x[i] < pivotValue:
            border += 1
            x[i], x[border] = x[border], x[i]
    x[low], x[border] = x[border], x[low]

    return (border)
for _ in range(n):
    array = np.random.randint(100, size=100)
    quisort (array)
qend = time.time()
qstart1 = time.time()
for _ in range(n):
    array1 = np.random.randint(100, size=500)
    quisort (array1)
qend1 = time.time()
qstart2 = time.time()
for _ in range(n):
    array2 = np.random.randint(100, size=1000)
    quisort (array2)
qend2 = time.time()
qstart3 = time.time()
for _ in range(n):
    array3 = np.random.randint(100, size=2000)
    quisort (array3)
qend3 = time.time()
qstart4 = time.time()
for _ in range(n):
    array4 = np.random.randint(100, size=5000)
    quisort (array4)
qend4 = time.time()
qstart5 = time.time()
for _ in range(n):
```

```
    array5 = np.random.randint(100, size=10000)
    quisort (array5)
qend5 = time.time()
```

Counting Sort

Counting Sort works best with lists that have a narrow range of elements.

The algorithm first determines the greatest value in the array, say k, and initialises a range of 0 to k.

Then it initialies a count of the range based on the number of instances of each element of the range in the array.

Then each element in the count array is added to the next and this gives the maximum position in the sorted list for each element in the original list. If there is more than one instance of an element value, it is placed in the position imediately before it in the sorted list. This iterates for each element in the original list until the sorted list is complete.

Counting Sort has best case efficiency of n+k, worst case efficiency of n+k and average case of n+k, a space complexity of n+k and is a Stable algorithm.

https://en.wikipedia.org/wiki/Counting_sort (https://en.wikipedia.org/wiki/Counting_sort)

https://www.geeksforgeeks.org/counting-sort/ (https://www.geeksforgeeks.org/counting-sort/)

https://www.youtube.com/watch?v=OKd534EWcdk&t=11s (https://www.youtube.com/watch?v=OKd534EWcdk&t=11s)

In [91]:

```
# Define the counting sort algorithm
cstart = time.time()
n = 10 # the function is to run 10 times
def cousort(x):
    mx = max(x)
    nl = [0 for i in range(mx+1)]
    rl = []
    for i in x:
        nl[i] += 1
    for i in range(len(nl)):
        if nl[i] != 0:
            rl += [i for n in range(nl[i])]
    return rl
for _ in range(n):
    array = np.random.randint(100, size=100)
    cousort (array)
cend = time.time()
cstart1 = time.time()
for _ in range(n):
    array1 = np.random.randint(100, size=500)
    cousort (array1)
cend1 = time.time()
cstart2 = time.time()
for _ in range(n):
    array2 = np.random.randint(100, size=1000)
    cousort (array2)
cend2 = time.time()
cstart3 = time.time()
for _ in range(n):
    array3 = np.random.randint(100, size=2000)
    cousort (array3)
cend3 = time.time()
cstart4 = time.time()
for _ in range(n):
    array4 = np.random.randint(100, size=5000)
    cousort (array4)
cend4 = time.time()
cstart5 = time.time()
for _ in range(n):
    array5 = np.random.randint(100, size=10000)
    cousort (array5)
cend5 = time.time()
```

In [95]:

```
times = [["Size", "100", "500", "1,000", "2,000", "5,000", "10,000"],["Bubble Sort", (b
end-bstart)/n, (bend1-bstart1)/n, (bend2-bstart2)/n, (bend3-bstart3)/n, (bend4-bstart4)
/n, (bend5-bstart5)/n],["Selection Sort", (send-sstart)/n, (send1-sstart1)/n, (send2-ss
tart2)/n, (send3-sstart3)/n, (send4-sstart4)/n, (send5-sstart5)/n],["Insertion Sort", (
iend-istart)/n, (iend1-istart1)/n, (iend2-istart2)/n, (iend3-istart3)/n, (iend4-istart4
)/n, (iend5-istart5)/n],["Quick Sort", (qend-qstart)/n, (qend1-qstart1)/n, (qend2-qstar
t2)/n, (qend3-qstart3)/n, (qend4-qstart4)/n, (qend5-qstart5)/n],["Counting Sort", (cend
-cstart)/n, (cend1-cstart1)/n, (cend2-cstart2)/n, (cend3-cstart3)/n, (cend4-cstart4)/n,
(cend5-cstart5)/n]]
```

In [96]:

```
dash ='-' * 67

for i in range(len(times)):
    if i == 0:
        print(dash)
        print('{:<20s}{:<8s}{:<8s}{:<8s}{:<8s}{:<8s}{:<8s}'.format(times[i][0],times[i]
[1],times[i][2],times[i][3],times[i][4],times[i][5],times[i][6]))
        print(dash)
    else:
        print('{:<20s}{:<8.3f}{:<8.3f}{:<8.3f}{:<8.3f}{:<8.3f}{:<8.3f}'.format(times[i]
[0],times[i][1],times[i][2],times[i][3],times[i][4],times[i][5],times[i][6]))
```

```
-------------------------------------------------------------------
Size                100     500     1,000   2,000   5,000   10,000
-------------------------------------------------------------------
Bubble Sort         0.002   0.066   0.304   1.086   8.444   30.030
Selection Sort      0.002   0.041   0.145   0.690   3.893   15.483
Insertion Sort      0.001   0.039   0.137   0.583   3.447   12.304
Quick Sort          0.000   0.003   0.006   0.013   0.045   0.129
Counting Sort       0.000   0.000   0.000   0.007   0.001   0.003
```

In [97]:

```
# to graph the results

df = pd.DataFrame(data=times[1:,1:],    # values
...               index=times[1:,0],    # 1st column as index
...               columns=times[0,1:])  # 1st row as the column names
print(df)
```

```
  File "<ipython-input-97-180e93e1e96a>", line 4
    ...               index=times[1:,0],    # 1st column as index
                          ^
SyntaxError: invalid syntax
```