Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

LAB-1

[Course : COMP 314]

Submitted by

Sudip Bhattarai

Roll no:11

Submitted to

Dr. Rajani Chulyadyo

April 18, 2024

Selection sort

This is the efficient sorting algorithm in which we need the array is divided into two parted the sorted and unsorted.Initially, the sorted section is empty, and the unsorted section contains the entire list.At each iteration the smallest number is chosen from unsorted section and placed right of sorted section i.e at each iteration the sorted section size increases end after sorted section index equals to size of an array.Due to two nested loop the time complexity of selection sort is $O(n^2)$.

Its implementation can be seen in the below code snippet.

```
1 def selection_sort(arr):
2     n = len(arr)
3     for i in range(n):
4         min_index = i
5         for j in range(i+1, n):
6             if arr[j] < arr[min_index]:
7                 min_index = j
8         swap(arr, i, min_index)
9     return arr;
10
11 def swap(arr, i, j):
12     arr[i], arr[j] = arr[j], arr[i]
13
14
```

Here we have repeatedly swapped the min index of i with the smallest number searched in each iteration using the swap function defined at line 11.

Insertion Sort

Insertion sort is similar to selection sort but in this case we consider the element of the first index to be smallest i.e key .At first iteration no comparison is to be made so we take index 0 and sorted list and remaining as unsorted . Then at each iteration we take the smallest index of unsorted and then compare it with the sorted list element until the correct position is found.The time complexity of insertion sort is $O(n^2)$.

The implementation of insertion sort is shown below in the code snippet.

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j ≥ 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
    return arr;
```

Test Case

Here we have written the text cases for both sorting algorithm which include the 4 different cases:

- Best case : Already sorted i.e. ascending order
- Word case: Reverse list i.e. descending order
- Normal Case: Random list
- Empty case: Empty list

```python
import unittest
from selectionsort import selection_sort;

class TestInsertion(unittest.TestCase):

    def test_descending(self):
        input_data=[5,4,3,2,1]
        sortedarray=selection_sort(input_data)
        self.assertListEqual(sortedarray,[1,2,3,4,5])

    def test_ascending(self):
        input_data=[1,2,3,4,5]
        sortedarray=selection_sort(input_data)
        self.assertListEqual(sortedarray,[1,2,3,4,5])

    def test_empty(self):
        input_data=[];
        sortedarray=selection_sort(input_data)
        self.assertListEqual(sortedarray,[])

    def test_sorting(self):
        input_data=[1,10,5,2,3,8,6,9];
        sortedarray=selection_sort(input_data)
        self.assertListEqual(sortedarray,
[1,2,3,5,6,8,9,10])
if __name__=='__main__':
    unittest.main()
```

```python
import unittest
from insertion import insertion_sort;

class TestInsertion(unittest.TestCase):

    def test_descending(self):
        input_data=[5,4,3,2,1]
        sortedarray=insertion_sort(input_data)
        self.assertListEqual(sortedarray,[1,2,3,4,5])

    def test_ascending(self):
        input_data=[1,2,3,4,5]
        sortedarray=insertion_sort(input_data)
        self.assertListEqual(sortedarray,[1,2,3,4,5])

    def test_empty(self):
        input_data=[];
        sortedarray=insertion_sort(input_data)
        self.assertListEqual(sortedarray,[])

    def test_sorting(self):
        input_data=[1,10,5,2,3,8,6,9];
        sortedarray=insertion_sort(input_data)
        self.assertListEqual(sortedarray,
[1,2,3,5,6,8,9,10])
if __name__=='__main__':
    unittest.main()
```

Test Result :

Here on running both of the file we have successfully passed all the test and the output can be seen below

```
(.venv) sudipbhattarai@Sudips-MacBook-Pro Algorithm lab % python test_insertion.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
(.venv) sudipbhattarai@Sudips-MacBook-Pro Algorithm lab % python test_selection.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Graph (Time vs Array size)

Here we have plotted the graph for complexity of both insertion sort and selection sort for array of size increasing 10 at each iteration until the maximum array size. At each iteration the time is calculated before and after sorting and its difference is recorded in an array .
The code snippet can be seen below

```python
from insertion import insertion_sort
from selectionsort import selection_sort
from timecalculation import current_time,cal_time
from randomarray import generate_random_array
import matplotlib.pyplot as plt
start_time = current_time()
time_array_insertion = []
increment_size = 10
time_array_selection = []
MAX_ARRAY_SIZE=500
for i in range(MAX_ARRAY_SIZE):
    increment_size += 10
    array = generate_random_array(0,increment_size)
    start_time = current_time()
    insertion_sort(array)
    end_time = current_time()
    time_array_insertion.append(cal_time(start_time, end_time))
    start_time = current_time()
    selection_sort(array)
    end_time = current_time()
    time_array_selection.append(cal_time(start_time, end_time))
plt.plot(time_array_insertion, label='Insertion Sort')
plt.plot(time_array_selection, label='Selection Sort')
plt.xlabel('Array Size(array size)')
plt.ylabel('Time(Second)')
plt.title('Comparison of Sorting Algorithms')
plt.legend()
plt.show()
```

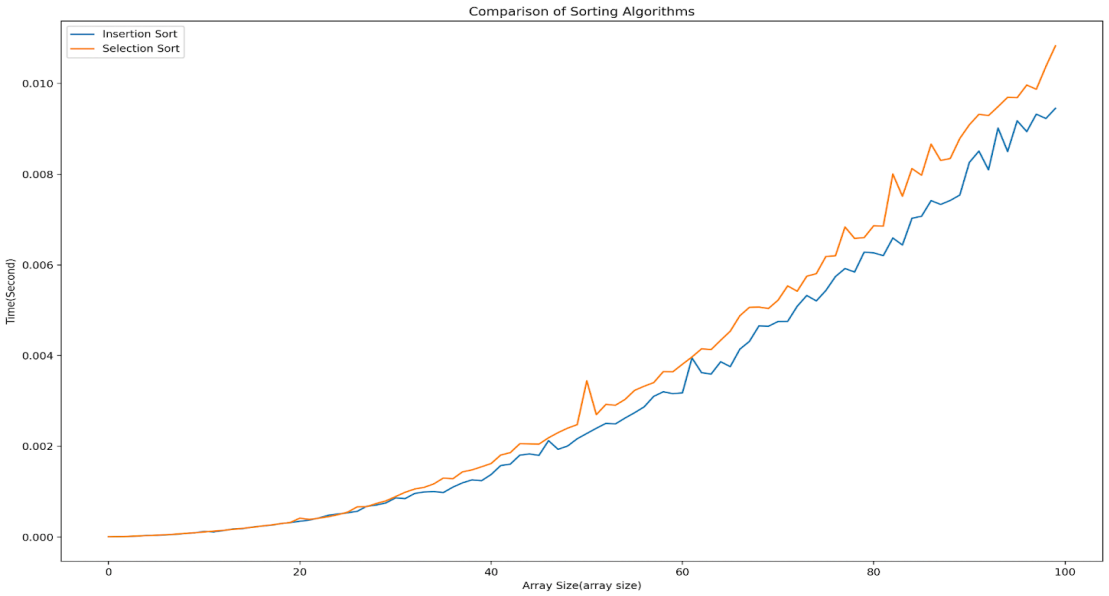
```python
import time
def current_time():
    return time.time()
def cal_time(start_time,end_time):
    result=end_time-start_time;
    return result


```

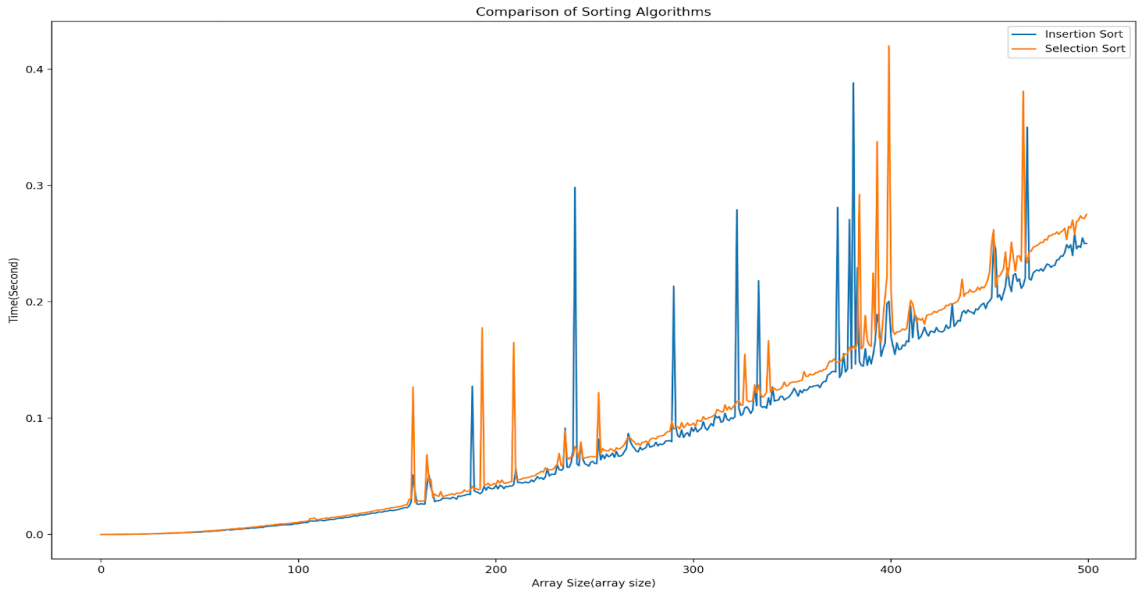
```
import random
def generate_random_array(start_range, end_range):
    if end_range == 0 or start_range >= end_range:
        return []
    return [random.randint(start_range, end_range) for _ in range(end_range-
start_range)]
  return go(f, seed, [])
}
```

The graph then obtained can be seen below for different range of array size

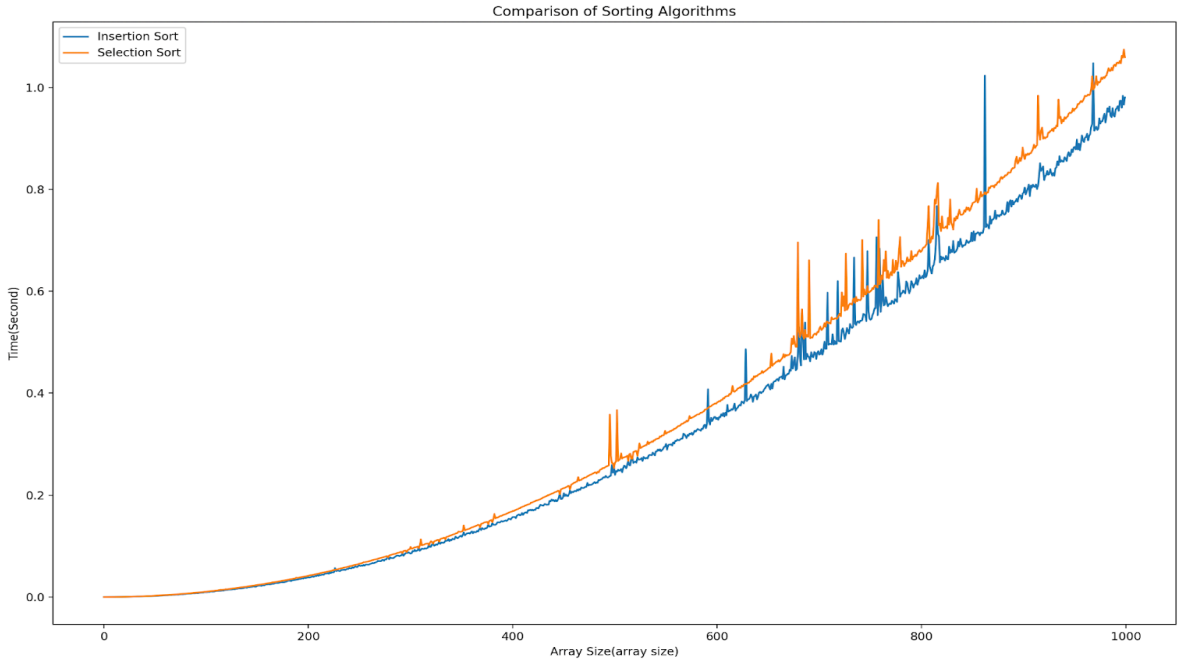
- Max Array size (100)



- Max Array Size (500)



- Max Array Size(1000)



From above graph we can observe that the curve of insertion sort and selection sort overlaps (takes the same amount of time to sort) but as the size of an array increases the time to sort varies Selection sort took comparatively more time then insertion sort to do sorting in the same data whereas insertion sort took less time than selection sort .As the best case time complexity of selection sort is $O(n^2)$ whereas the best case and worst case time complexity of insertion sort is $O(n)\ and\ O(n^2)$. So ,on comparing this two sorting algorithm we can see the insertion sort is more preferred than selection sort.