Assignment 2

Jasindan Rasalingam	**100584595**
Darron Singh	**100584624**
Ashwin Kamalakannan	**100584423**

Link to repository -> https://github.com/AshwinK97/Algorithms/tree/master/Assignment%202

Pseudo Algorithm

We randomly generate a set of 20 (x, y) values. We pass these values to the paretofront() function along with the mode. The mode is used to check which pareto front set we want.

1. Low values for u and v are desirable
2. High values for u and v are desirable
3. Low values for u and high values for v are desirable
4. High values for u and low values for v are desirable

For **case 1**, we sort the points based on ascending y value then x value. We then store the first point as the front. Then, we iterate through each of the sorted points and we check if the x values are greater than our front point. If it is, we change front to the new point and add it to the pareto set. Then we return the pareto set and plot the points, saved to 'output1.png'.

```
points = sort([u, v] by v then u)
front = points[0]
pareto = []
for point in points:
	if point[x] >= front[x]:
		front = point
		pareto append front
return pareto
```

For **case 2**, we sort the points based on descending x value then y value. We then store the first point as the front. Then, we iterate through each of the sorted points and we check if the y values are greater than or equal to our front point. If it is, we change front to the new point and add it to the pareto set. Then we return the pareto set and plot the points, saved to 'output2.png'.

```
points = sort([u, v] by u then v)
front = points[0]
pareto = []
for point in points:
	if point[y] >= front[y]:
		front = point
		pareto append front
return pareto
```

For **case 3**, we sort the points based on ascending y value then x value. We then store the first point as the front. Then, we iterate through each of the sorted points and we check if the x values are less than or equal to our front point. If it is, we change front to the new point and add it to the pareto set. Then we return the pareto set and plot the points, saved to 'output3.png'.

```
points = sort([u, v] by v then u)
front = points[0]
pareto = []
for point in points:
	if point[x] <= front[x]:
		front = point
		pareto append front
return pareto
```

For **case 4**, we sort the points based on ascending y value then x value. We then store the first point as the front. Then, we iterate through each of the sorted points and we check if the y values are greater than or equal to our front point. If it is, we change front to the new point and add it to the pareto set. Then we return the pareto set and plot the points, saved to 'output3.png'.

```
points = sort([u, v] by u then v)
front = points[0]
pareto = []
for point in points:
	if point[y] >= front[y]:
		front = point
		pareto append front
return pareto
```

All four algorithms have the exact same time complexity since we are not adding any other lines of code. For each of the 4 algorithms, we are using pythons built in sort function which uses an implementation of the Tim sort algorithm. This sorting algorithm is a combination of the insertion and merge sort algorithms. This means that the overall time complexity of our algorithm will be dominated by the merge sort nlogn run time.

We used the 'mode' variable to tell the paretofront() function which case we wanted to find. This allowed us to put all 4 algorithms in a single function with if statements branching to them. Our export plot function uses the matplotlib library to plot the points as a scatter plot, and then the pareto set as a broken line plot.

Runtime Analysis

We used Python 2.7 running on a custom desktop PC with an AMD FX-8320 CPU @3.5GHz and 16GB RAM, running Ubuntu 14.04 in Windows subsystem for Linux.

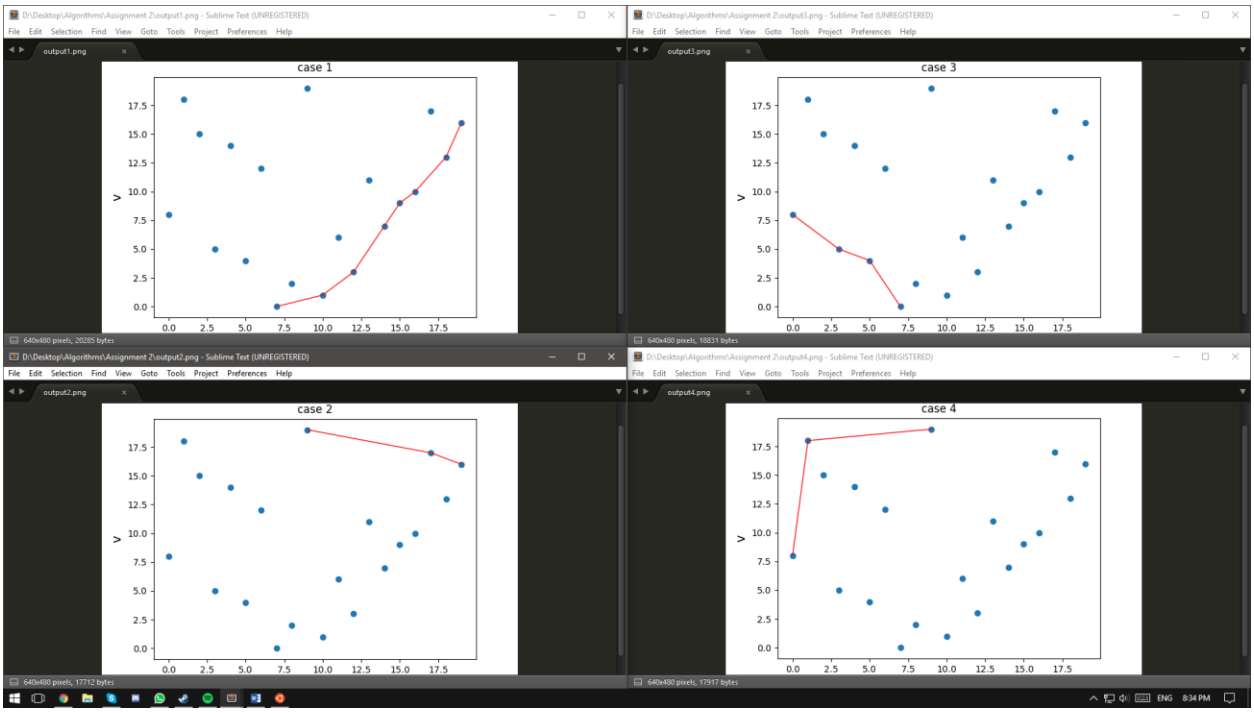
Our algorithm runs in O(n*log(n)) time. Even though we are only using a single for loop in our algorithm, we also use the built-in python sort function. This sort function is called Tim sort and it is a hybrid of merge sort and insertion sort. Because of this, our algorithm is dominated by the merge sort algorithm and our overall time complexity is nlogn. One viable way to improve our algorithm's run time is to use radix sort or counting sort instead of merge sort. This is because radix sort has a worst case of

O(n*k) and counting sort has a worst case of O(n+k). By changing to either of these sorting algorithms, we could essentially have a linear runtime.

Test Cases

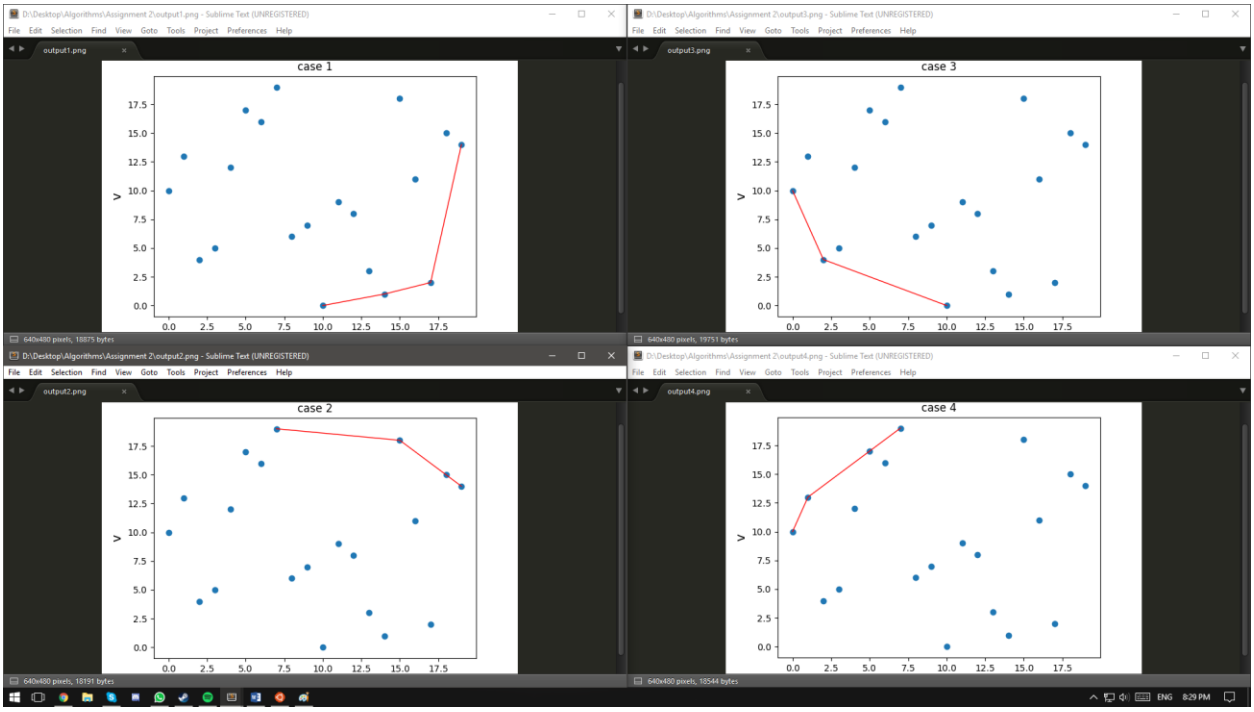
Example 1

```
Ashwin@DESKTOP-QMO41AP:/mnt/d/Desktop/Algorithms/Assignment 2$ python solution.py
points: [(8, 2), (5, 4), (4, 14), (0, 8), (19, 16), (14, 7), (15, 9), (12, 3), (11, 6), (7, 0), (3,
 5), (9, 19), (18, 13), (13, 11), (1, 18), (17, 17), (2, 15), (16, 10), (6, 12), (10, 1)]
low u low v: [(7, 0), (10, 1), (12, 3), (14, 7), (15, 9), (16, 10), (18, 13), (19, 16)] 5.6e-05 sec
onds
high u high v: [(19, 16), (17, 17), (9, 19)] 3e-05 seconds
low u high v: [(7, 0), (5, 4), (3, 5), (0, 8)] 2.6e-05 seconds
high u low v: [(0, 8), (1, 18), (9, 19)] 2.8e-05 seconds
Ashwin@DESKTOP-QMO41AP:/mnt/d/Desktop/Algorithms/Assignment 2$
```



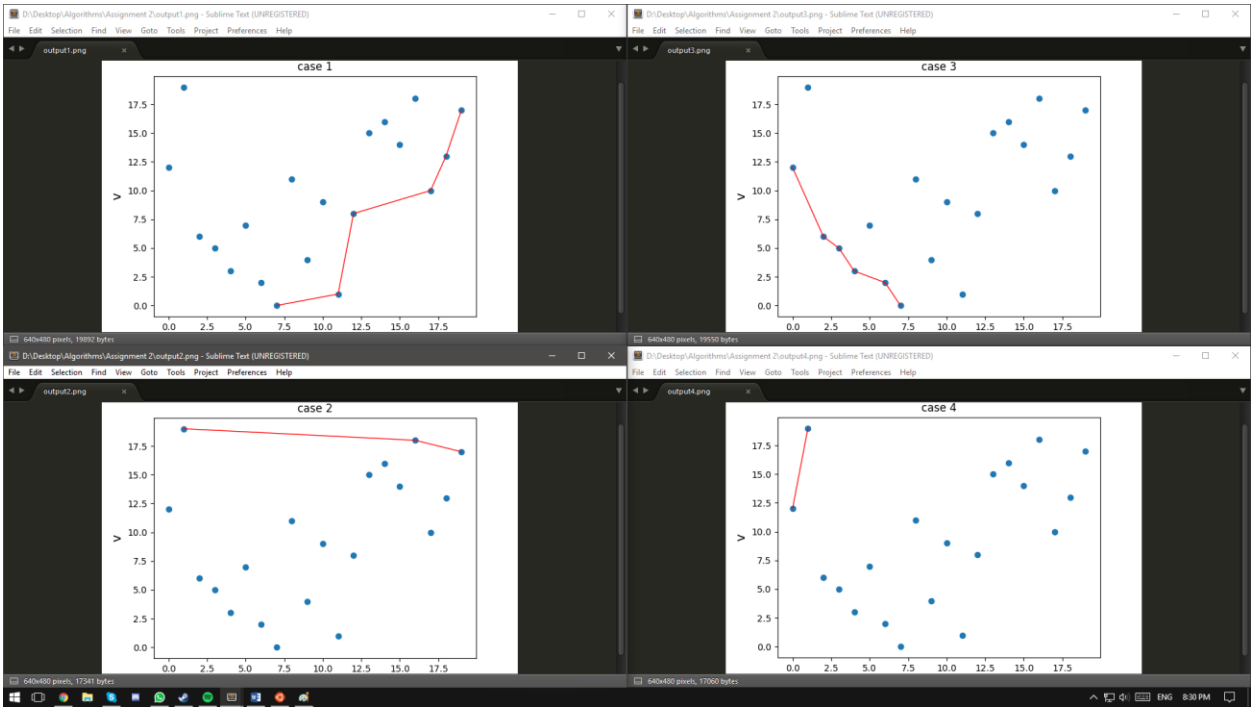
Example 2

```
Ashwin@DESKTOP-QMO41AP:/mnt/d/Desktop/Algorithms/Assignment 2$ python solution.py
points: [(0, 10), (3, 5), (18, 15), (2, 4), (10, 0), (8, 6), (12, 8), (7, 19), (14, 1), (11, 9), (1
9, 14), (9, 7), (5, 17), (13, 3), (6, 16), (4, 12), (1, 13), (16, 11), (17, 2), (15, 18)]
low u low v: [(10, 0), (14, 1), (17, 2), (19, 14)] 2.8e-05 seconds
high u high v: [(19, 14), (18, 15), (15, 18), (7, 19)] 3e-05 seconds
low u high v: [(10, 0), (2, 4), (0, 10)] 3.6e-05 seconds
high u low v: [(0, 10), (1, 13), (5, 17), (7, 19)] 2.9e-05 seconds
Ashwin@DESKTOP-QMO41AP:/mnt/d/Desktop/Algorithms/Assignment 2$
```



Example 3

```
Ashwin@DESKTOP-QMO41AP:/mnt/d/Desktop/Algorithms/Assignment 2$ python solution.py
points: [(7, 0), (5, 7), (8, 11), (9, 4), (1, 19), (10, 9), (16, 18), (14, 16), (18, 13), (17, 10),
 (6, 2), (2, 6), (4, 3), (13, 15), (19, 17), (11, 1), (12, 8), (3, 5), (0, 12), (15, 14)]
low u low v: [(7, 0), (11, 1), (12, 8), (17, 10), (18, 13), (19, 17)] 5.4e-05 seconds
high u high v: [(19, 17), (16, 18), (1, 19)] 2.8e-05 seconds
low u high v: [(7, 0), (6, 2), (4, 3), (3, 5), (2, 6), (0, 12)] 2.7e-05 seconds
high u low v: [(0, 12), (1, 19)] 2.6e-05 seconds
```



Example 4

```
Ashwin@DESKTOP-QMO41AP:/mnt/d/Desktop/Algorithms/Assignment 2$ python solution.py
points: [(3, 8), (0, 12), (8, 0), (1, 6), (19, 2), (10, 4), (14, 10), (11, 13), (4, 1), (2, 5), (6,
 7), (16, 3), (7, 19), (18, 15), (17, 11), (5, 17), (9, 9), (15, 16), (12, 18), (13, 14)]
low u low v: [(8, 0), (19, 2)] 5.5e-05 seconds
high u high v: [(19, 2), (18, 15), (15, 16), (12, 18), (7, 19)] 2.9e-05 seconds
low u high v: [(8, 0), (4, 1), (2, 5), (1, 6), (0, 12)] 4e-05 seconds
high u low v: [(0, 12), (5, 17), (7, 19)] 3e-05 seconds
Ashwin@DESKTOP-QMO41AP:/mnt/d/Desktop/Algorithms/Assignment 2$
```