**A) Asymptotic Upper Bound for Quicksort**

I used randomized Quicksort for this homework. It has 2 main functions:

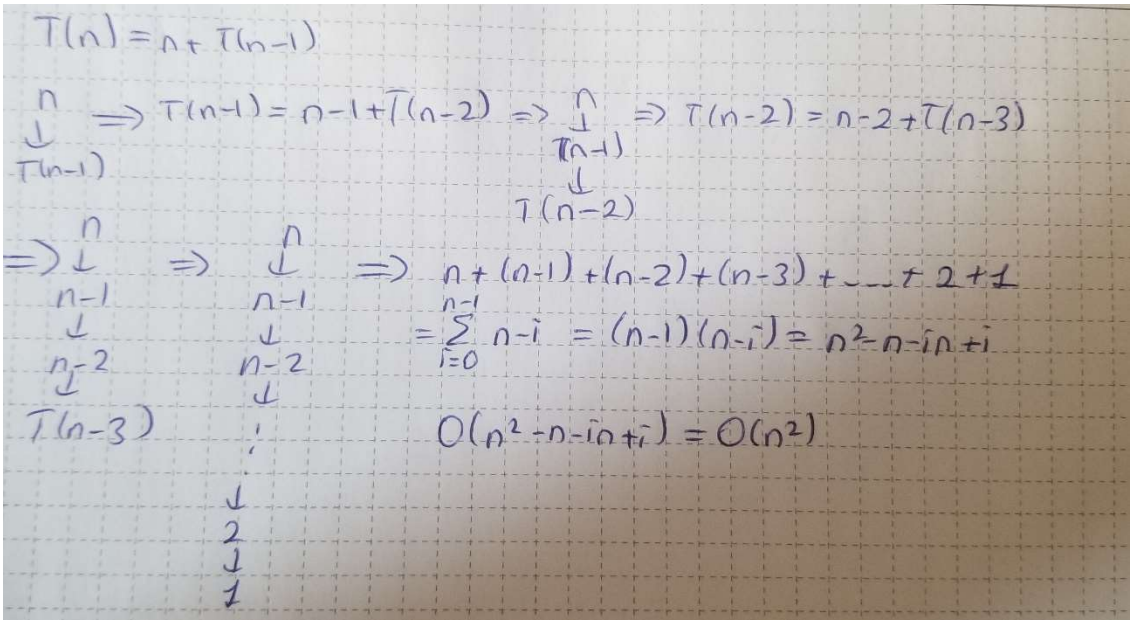
**Quicksort(low , high)**

```
Do if low<high
        randomPivot = partition_r(low, high)
        Quicksort(low, randomPivot-1)
        Quicksort(randomPivot+1,high)
```

**partition(low,high)**

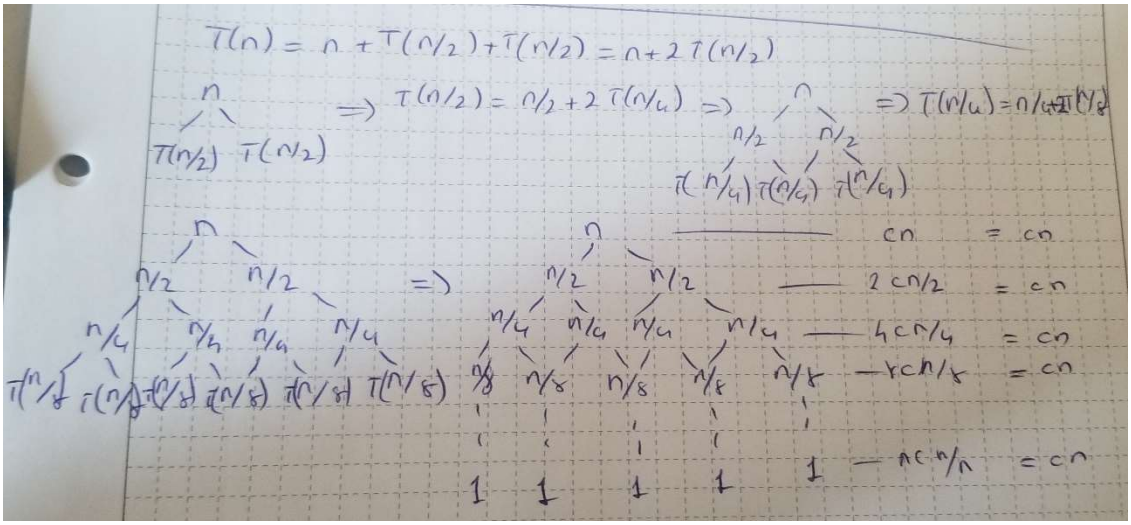
```
range = high - low
r = rand()%range + low
swap TimeStockList[r]<->TimeStockList[high]
pivot = TimeStockList[high]
i = low-1;
For j <-low to high
        A = pivot.compare(pivot, TimeStockList[j],criteria)
        Do if A <= 0 //if pivot is smaller or equal
                i++
                swap TimeStockList[i]<->TimeStockList[j]
swap TimeStockList[i+1] <->TimeStockList[high]
return i+1
```

**Worst case of Quicksort**



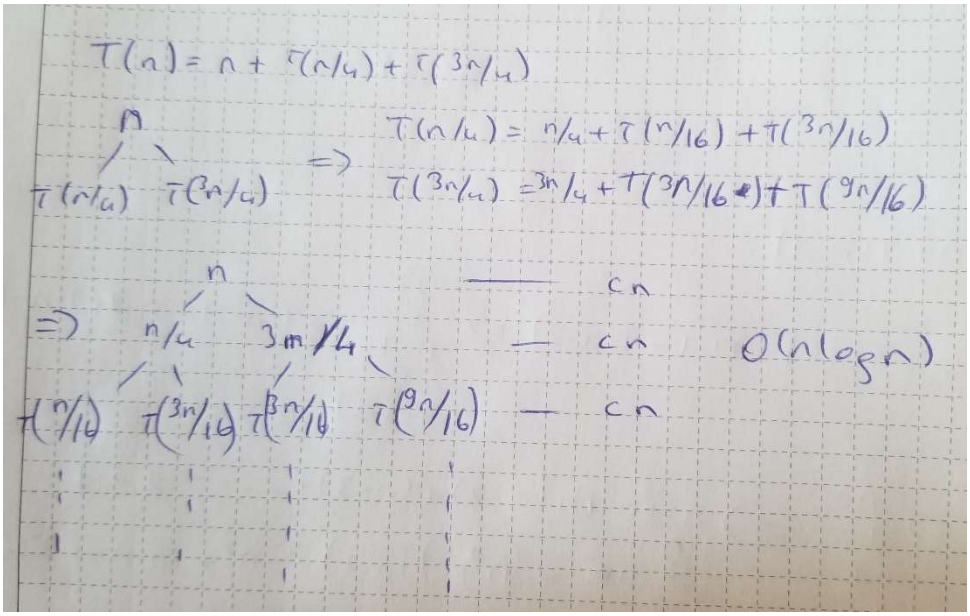

Quick sorts bound is $O(n^2)$ if the data is already sorted. Because the pivot is selected from the last element of the array and because all the remaining elements in the array is always biggest or the smallest, so it can only sort one element at each iteration, so it will call itself n-i times and look through the list which is length is also n-i.

**Best Case of Quicksort**




If we assume that we picked a pivot that is right in the middle of the array the asymptotic upper bound will be Θ(nlog2n) which is same as merge sort. So, the code will split the vector half each time it iterates. It will call itself with half of its length until it reaches a vector segment that has one element. So, we can say it will iterate log(n) times.

**Average Case for Quicksort**

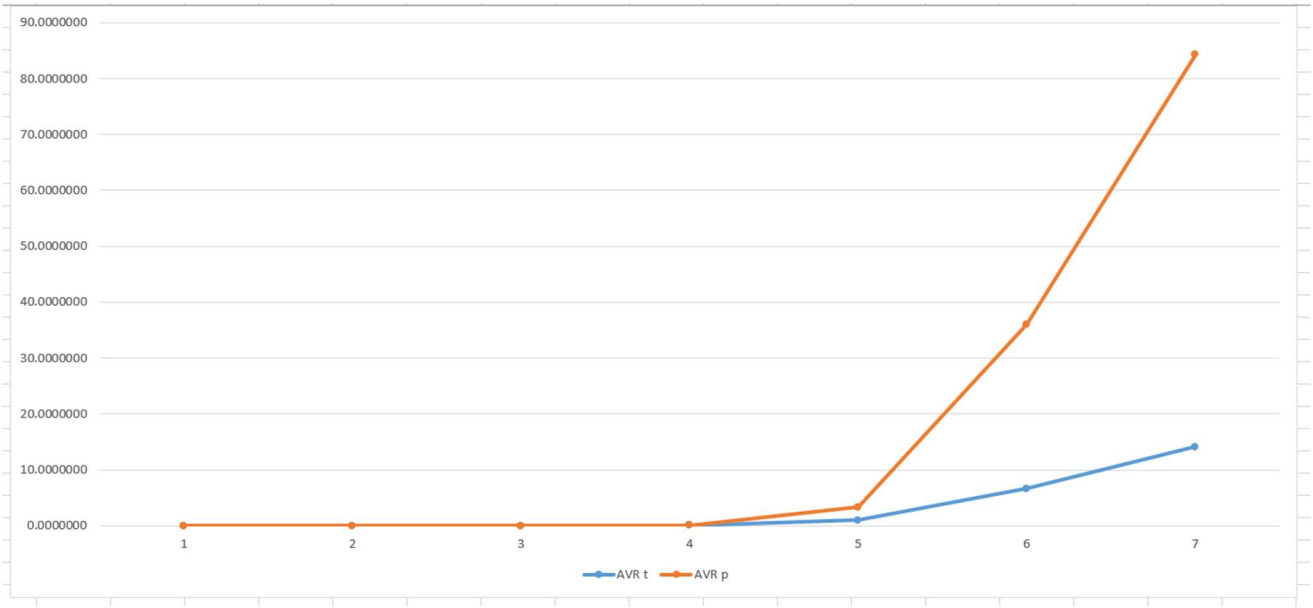



The average case for quicksort is O(nlogn), same as the best case but the constant in front of it a little bit bigger.

**B)**

| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | | 10.0000000 | 100 | 1000 | 10000 | 100000 | 500000 | 1000000 |
| 2 | | 0.0000130 | 0.000391 | 0.003144 | 0.059395 | 1.12514 | 7.90236 | 15.7005 |
| 3 | | 0.0000130 | 0.000182 | 0.003086 | 0.044281 | 1.04151 | 6.33832 | 13.9392 |
| 4 | | 0.0000120 | 0.00021 | 0.003143 | 0.049692 | 0.975505 | 6.22617 | 13.5013 |
| 5 | | 0.0000140 | 0.000178 | 0.003333 | 0.045662 | 0.988117 | 7.02409 | 13.9473 |
| 6 | | 0.0000130 | 0.000248 | 0.003222 | 0.044038 | 1.07699 | 6.50408 | 13.2959 |
| 7 | | 0.0000120 | 0.000187 | 0.003129 | 0.050542 | 0.942702 | 6.0957 | 14.2787 |
| 8 | | 0.0000130 | 0.0002 | 0.003262 | 0.046351 | 0.977577 | 6.8186 | 13.9757 |
| 9 | | 0.0000130 | 0.000211 | 0.003344 | 0.052667 | 0.931882 | 6.33953 | 14.7441 |
| 10 | | 0.0000130 | 0.000287 | 0.003245 | 0.052583 | 0.978497 | 6.35257 | 13.9724 |
| 11 | t | 0.0000160 | 0.000211 | 0.003392 | 0.053519 | 1.02077 | 6.85661 | 13.4143 |
| 12 | AVR t | 0.0000132 | 0.000231 | 0.003230 | 0.049873 | 1.00587 | 6.646 | 14.07694 |
| 13 | | 0.000010 | 0.000175 | 0.003721 | 0.078948 | 3.24992 | 36.1642 | 87.6914 |
| 14 | | 0.000010 | 0.000171 | 0.003466 | 0.082874 | 3.16549 | 33.752 | 80.755 |
| 15 | | 0.000011 | 0.000172 | 0.003703 | 0.079945 | 3.60388 | 34.5934 | 82.7415 |
| 16 | | 0.000010 | 0.000156 | 0.003382 | 0.078908 | 3.34309 | 35.9773 | 86.65 |
| 17 | | 0.000011 | 0.000153 | 0.00337 | 0.079089 | 3.32451 | 35.1555 | 83.9872 |
| 18 | | 0.000011 | 0.000175 | 0.003524 | 0.079415 | 3.29405 | 36.2351 | 87.9075 |
| 19 | | 0.000008 | 0.000175 | 0.003132 | 0.077557 | 3.36606 | 36.1625 | 81.357 |
| 20 | | 0.000009 | 0.000175 | 0.003223 | 0.07887 | 3.35627 | 35.0791 | 85.3782 |
| 21 | | 0.000011 | 0.000193 | 0.003439 | 0.080039 | 3.19617 | 39.3273 | 83.967 |
| 22 | p | 0.000009 | 0.000193 | 0.003409 | 0.077302 | 3.39099 | 38.3164 | 82.0916 |
| 23 | AVR p | 0.0000100 | 0.000174 | 0.003437 | 0.079295 | 3.32904 | 36.076 | 84.3 |

Data and the average case for time stamp = t and last price = p

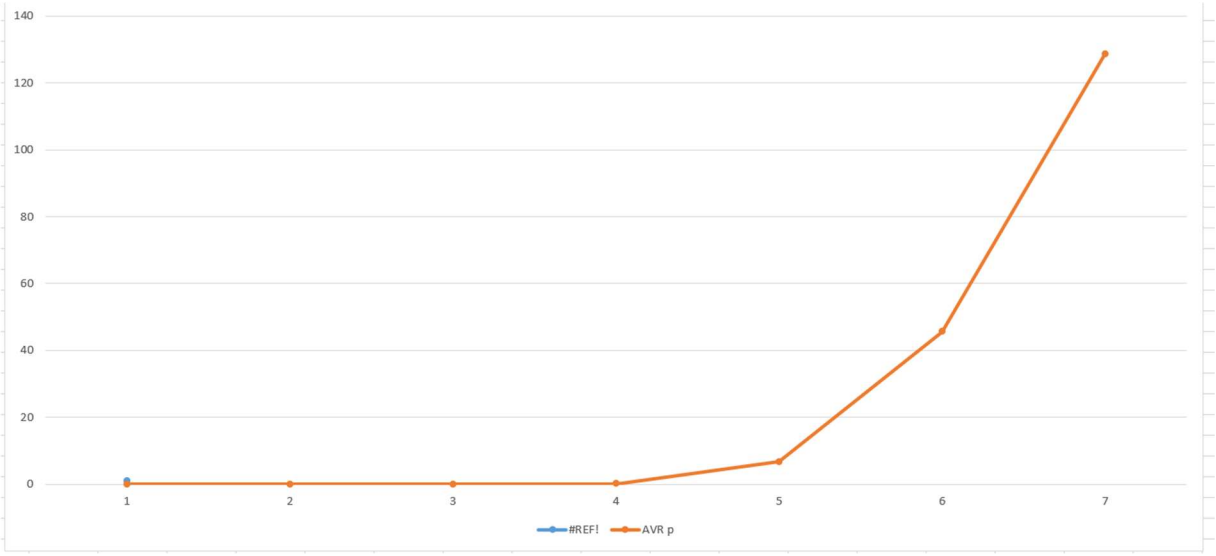



The two-line chart for the average time for time stamp and last price.

Because, the time stamp was reversed sorted it sorted faster than price. The sort time of time stamp is really close to O(n) and time of sorting price still smaller than $O(n^2)$ which is the wort case.

**C) A**

| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | | 10 | 100 | 1000 | 10000 | 100000 | 500000 | 1000000 |
| 2 | | 0.000013 | 0.000301 | 0.009824 | 0.185872 | 6.52968 | 46.4477 | 124.921 |
| 3 | | 0.000013 | 0.000303 | 0.009082 | 0.179869 | 6.69901 | 46.5022 | 126.872 |
| 4 | | 0.000012 | 0.000305 | 0.009213 | 0.178264 | 6.92614 | 46.1575 | 135.74 |
| 5 | | 0.000013 | 0.000317 | 0.008926 | 0.188551 | 6.95816 | 44.2543 | 119.962 |
| 6 | | 0.000013 | 0.000315 | 0.008837 | 0.180485 | 6.86107 | 44.8796 | 127.643 |
| 7 | | 0.000013 | 0.000302 | 0.00972 | 0.17951 | 6.8389 | 46.768 | 138.643 |
| 8 | | 0.000014 | 0.000342 | 0.009582 | 0.178388 | 6.7289 | 45.4976 | 128.53 |
| 9 | | 0.000014 | 0.000299 | 0.008886 | 0.175447 | 7.0222 | 44.3531 | 129.8237 |
| 10 | | 0.000013 | 0.000306 | 0.009021 | 0.184254 | 6.39187 | 45.5705 | 126.937 |
| 11 | p | 0.000015 | 0.000317 | 0.00912 | 0.181932 | 6.53484 | 45.0426 | 127.353 |
| 12 | AVR p | 0.000013 | 0.000311 | 0.009221 | 0.181257 | 6.74908 | 45.547 | 128.642 |




We can see that the graph for price is now closer to $O(n^2)$ which is the worst case. The worst case happens when the array is sorted so when I run the algorithm on the sorted array the time for it get closer to the $O(n^2)$. To prevent this we can shuffle the array with:

"random_shuffle(rTimeStockList.begin(),rTimeStockList.end());"

This way we can make sure that the array is shuffled and not sorted.

Or we can just say do not change if the place is same as:

"if(i!=j){ swap(i,j);}"

This way we can prevent unneccesarry swap which also consumes time.

Or we can choose a random pivot which also helps to prevent to pick the largest or smallest number in set.

**D)** We change the place of the variable if it is equal so the algorithm of Quicksort is not stable.

"Do if A <= 0 //if pivot is smaller or equal"

For example if we sort a data set looks like [1,3,4,3,5] and if we choose the first 3 as a pivot the sequence to sort will be :

[1,**3**,4,3,5] -> [1,5,4,3,**3**] -> [1,3,**3**,5,4] -> … -> [1,3,**3**,4,5]

The first 3 is now the second and the second 3 is now the first. So, we can say that quicksort is not stable.