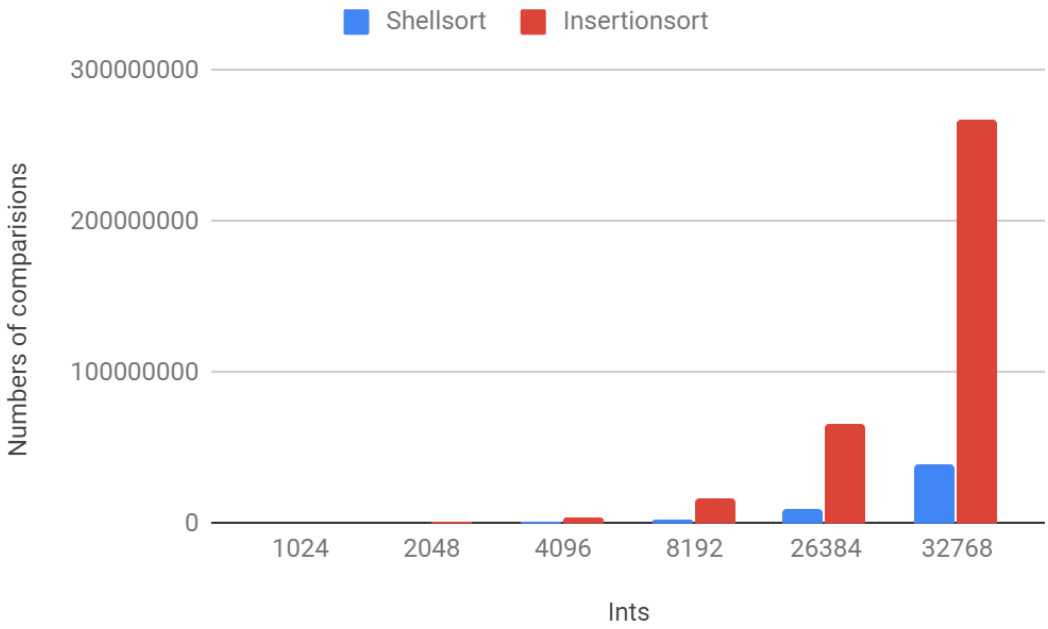
HW2
Yuhang Zhou (yz853)
2/25/2019

Q1.
Result

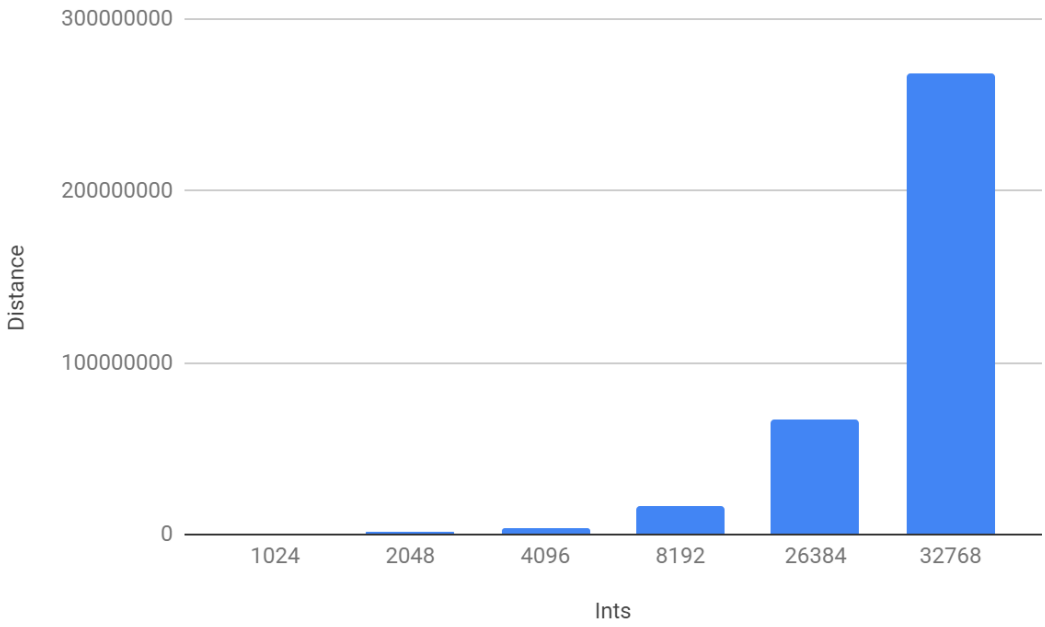
Ints	Shellsort	Insertionsort
1024	46768	265564
2048	169081	1029283
4096	660673	4187899
8192	2576322	16936958
26384	9950984	66657566
32768	39442505	267966675



In the plot, it's obvious that the shell sort is much better than insertion sort. With the increasing of the input number, the comparison number of insertion sort increases much faster than that in shell sort.
In the shell sort, in the beginning, the gap or the h is usually big, in this example, it starts from 7. So when the gap is 7, the number of comparison should be very small because the gap is big, so that in one sort loop, there are just n/7 data to compare and sort. Then the gap is 3, which also should be much better than gap 1. At last, the gap comes to 1, at this time, the whole data is almost sorted well, data is closed to all sorted, so in this case, the number of comparisons also should be small.
While in the insertion sort, the gap is 1 in the beginning. At this time, the data is really unorganized and messy which makes the gap 1 process very hard, so that the number of comparisons will be very big when compared with shell sort.

Q2

Ints	Distance
1024	264541
2048	1027236
4096	4183804
8192	16928767
26384	66641183
32768	267933908



Because the arrays from data0.* are all in order, so that the Kendall Tau distance between these two arrays is that the times of doing swap operations when I use insertion sort to sort the data1.* arrays.
Because the time complexity of insertion sort is unstable, but the worst case is O(n^2). So the average computation time of this algorithm is less than quadratic time.

Q3
Detail shown in codes hw2q3.cpp
The algorithm I designed for this problem is based on the Bottom-up merge sort algorithm which is in the Q4.
I watch this data set as blocks. In this sample, I watch it as 4 blocks, and data of each block is the same.
Firstly, I go through all the data, and record the index of the last one of each block, or the one before the first one of each block. In this sample, it's `{-1,1023,3071,7167,8191}`
After that, I merge the blocks, but it's a little different with the Bottom-up. For example, in

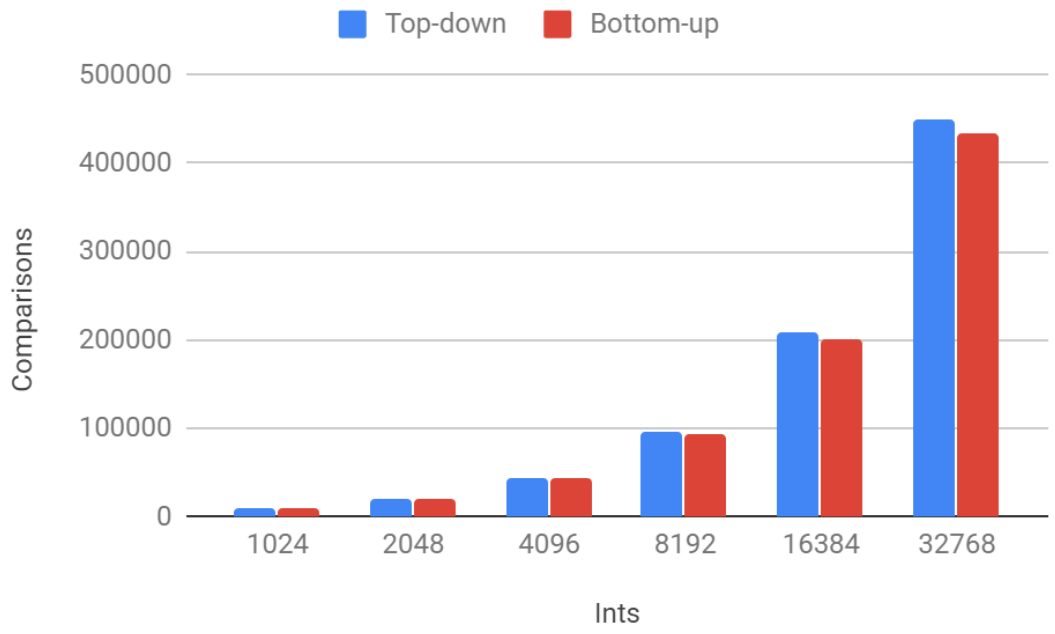
bottom up, the first merge is merge a[0] and a[1], while in this algorithm, I merge block 1 and block 2.

Then inside the merge() function, I change it a little bit. In the bottom up, the program compares two data from two parts, then put the smaller one in the final array. While what I did is that, after put this data in the array, I also take a look at the next data from this part, If the next data is the same, so that I can put the next data to array too. So that I the next data doesn't need to compare with the data of another part.

I think my algorithm will be efficient in this case. Because firstly it's based on bottom up merge sort, then I take advantage of the feature of the data, which is they repeat a lot. The bottom-up merge sort starts from merging one data and one data, while I starts from merging one block and one block. Also inside the merge() function, it saves lots of comparison because I put the data in the array if the data repeats, no more comparison.

Q4

Ints	Top-down	Bottom-up
1024	8954	8739
2048	19934	19380
4096	43944	43379
8192	96074	92967
16384	208695	199703
32768	450132	434073



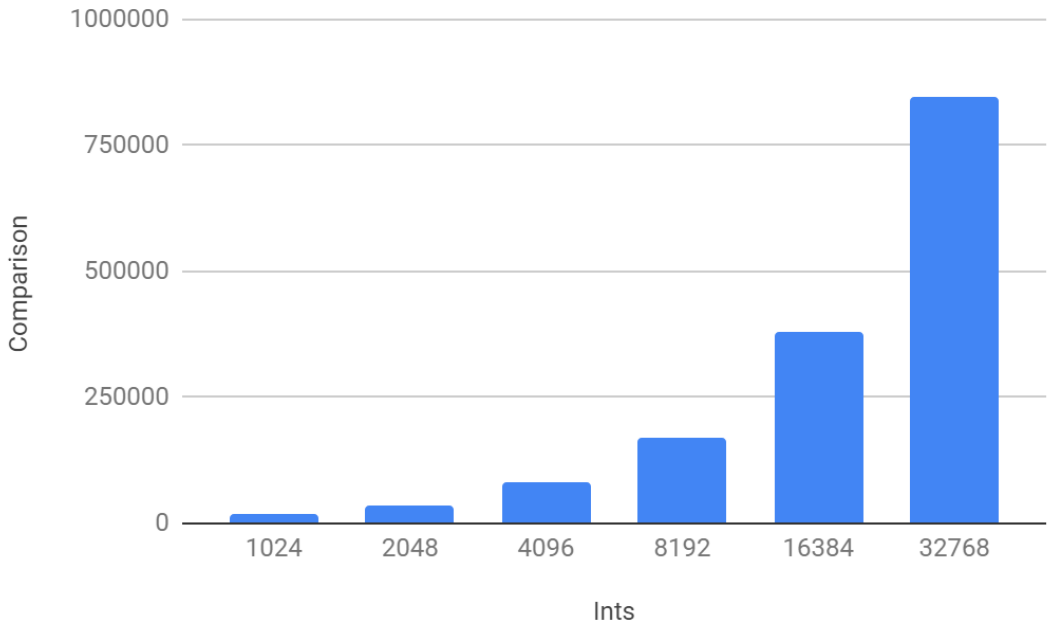
The result shows that the times of comparisons in "Top-down" and "Bottom-up" algorithms are almost the same. Because the top-down and the bottom-up is similar.

In the top-down algorithm, the program will divide the data set into 2 parts, then divide each

of these 2 parts into another smaller 2 parts. Lets say the number's of data is n, the n will be divided by 2, until the n=1. It's kind of like lg(n) in the binary research.
While in the bottom-up algorithm, the n starts from 1, then n=n*2, until two of n size parts can cover all the data. So the bottom-up is like a reverse version of the top-down algorithm. That's why the comparisons in these two algorithms are very similar.
After some computation, it's easy to see that the trend of the comparison times with the increasing of input data is proportional to n lg(n), which makes sense. Because times of comparison is the exactly the same as the times of the transferring operation a[ ]=aux[ ]. Every time the program do a comparison, one data will transfer from aux[ ] to a[ ]. Also we know that the complexity of merge sort is stable which is O(n lg(n)), that's why the comparison times also goes like n lg(n).

Q5

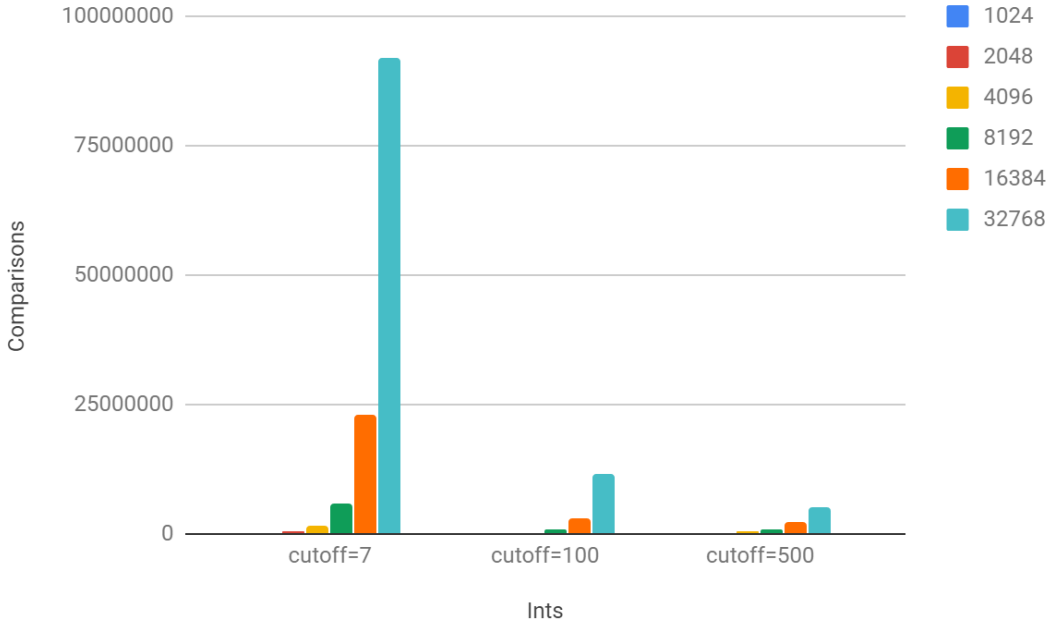
Ints	Comparison
1024	16252
2048	35750
4096	81466
8192	168621
16384	379095
32768	844255



Compared to the merge sort in Q4, the number of comparisons is about twice of that in merge sort.

After the cutoff is added, the results are shown below.

Ints	cutoff=7	cutoff=100	cutoff=500
1024	105077	36996	95956
2048	401551	93622	212912
4096	1528770	282550	401212
8192	5933401	890419	880635
16384	23039867	3118711	2156523
32768	91895922	11691162	5340492



When I set the cutoff=7, the efficiency is really bad compares to the case without cut off.
I test some different value for cutoff, and I never find one which is better than the ones without cutoff. But I find some good situation.
For 1024 input data, the cutoff=7 is almost the best case, while it's still twice of the original program. For 32768 inputs, the best case is around cutoff=500.