HACETTEPE UNIVERSITY

DEPARTMENT OF
COMPUTER ENGINEERING

BBM204 PROGRAMMING LAB. ASSIGNMENT 1
Analysis of Sorting Algorithms

Author
Kayla AKYÜZ
21726914
b21726914@cs.hacettepe.edu.tr

Advisor
Dr. Ahmet Selman BOZKIR
selman@cs.hacettepe.edu.tr

sorter.java

March 26,2020

Cocktail Shaker Sort Algorithm

With cocktail shaker sort algorithm we go front and back in our list and move along the biggest and smallest values we find in our way to their right place.

```java
static public long cocktail(List<Integer> A){
    long startTime = System.nanoTime();
    boolean swapped = true;
    int start = 0;
    int limit = A.size();
    while (swapped) {
        swapped = false;
        for(int i = start; i < limit - 1; i++){
            if(A.get(i)>A.get(i+1)){
                int temp = A.get(i);
                A.set(i,A.get(i+1));
                A.set(i+1,temp);
                swapped = true;
            }
        }
        if (!swapped)
            break;
        swapped = false;
        limit = limit - 1;
        for (int i = limit - 1; i >= start; i--) {
            if (A.get(i) > A.get(i+1)) {
                int temp = A.get(i);
                A.set(i, A.get(i+1));
                A.set(i+1,temp);
                swapped = true;
            }
        }
        start = start + 1;
    }
    long endTime   = System.nanoTime();
    return endTime - startTime;
}
```

Hypothesis

Time

Regardless of how elements in the list distributed, we have to go back and forth in order to place them. As long as there is a misplaced element this will trigger function to go all the way and check the rest of the list and come back checking again. No way of knowing if our misplaced item was fixed along the way. This brings us an average case time complexity of O(n^2). Worst case scenario is still same we have to check back and forth until everything is placed and this would bring us O(n^2). Unless the list is already sorted. This time when we are going forward we are never going to trigger any swap so when we reach the end we will not come back checking again. The algorithm knows it is already sorted. For best case it takes O(n) time complexity. This algorithm is perfect for checking if a given list is sorted or not but in terms of sorting an unsorted list it performs badly.

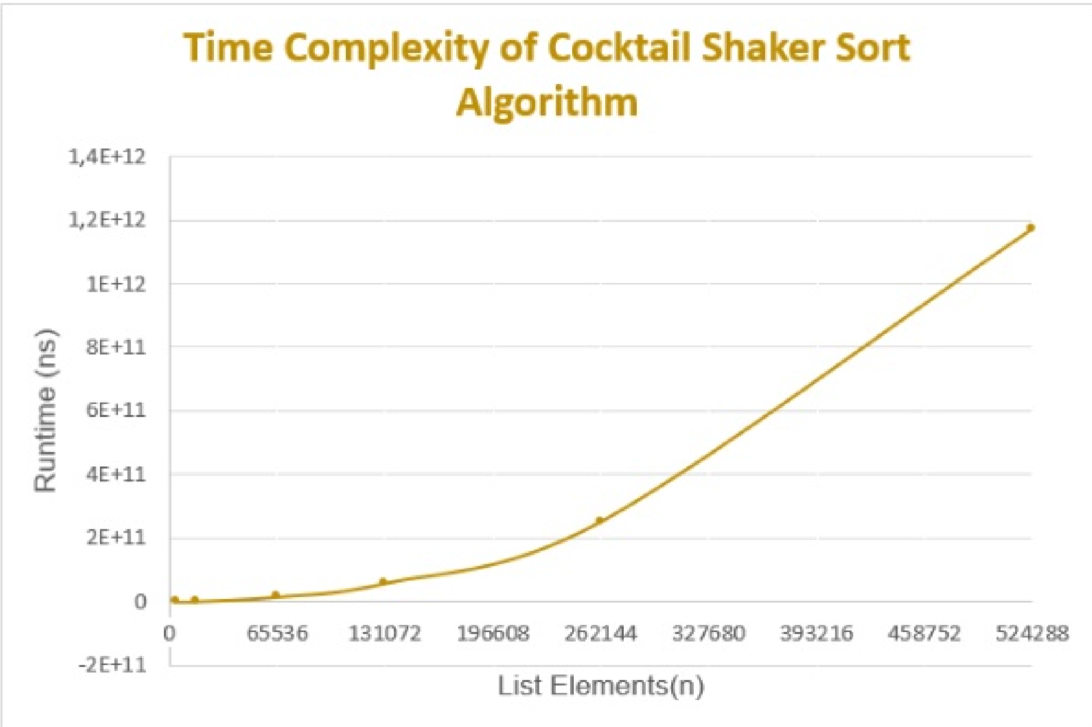
Space

Cocktail shaker algorithm only uses swap functions on given list so it takes no extra space other than temp memory used while swapping. This would make it's space complexity O(1).

Testing Results

Performed on same randomly generated integers of range 0-1147483647.

Cocktail Sort	Test No 1	Test No 2	Test No 3	Test No 4	Test No 5	Avarage
4096	72954801	49773000	52109601	51629600	51310300	55555460,4
16384	963381100	1031649000	1184311401	1171952700	1152068600	1100672560
65536	14695981800	15419549500	16277307801	15677560901	15403051700	15494690340
131072	59395755700	58932867999	58570098901	58713719299	58323203600	58787129100
262144	2,53236E+11	2,59174E+11	2,49973E+11	2,51199E+11	2,50209E+11	2,52758E+11
524288	1,24133E+12	1,20247E+12	1,14299E+12	1,14212E+12	1,14146E+12	1,17407E+12



Conclusion

By running time reaching approximately an hour for 524288 items cocktail sort is the least time efficient algorithm. The proof of O(n^2) average time complexity can be seen by looking at the curve of the chart. It seems this algorithm would be only useful to checking whatever a list is sorted because of it's time complexity.

Pigeonhole Sort Algorithm

With pigeonhole sort algorithm we create pigeon holes and put elements of our list in to according holes. The principle of hole creation lies with in the range of our list. After done we look at the holes and replace them accordingly in a sorted manner.

```java
static public long pigeon(List<Integer> A){
    long startTime = System.nanoTime();
    int min = A.get(0);
    int max = A.get(0);
    int range;
    for(int i = 1; i<A.size();i++){
        if(A.get(i)<min)
            min = A.get(i);
        if(A.get(i)>max)
            max = A.get(i);
    }
    range = max - min + 1;
    int[] holes = new int[range];
    for (Integer integer : A) holes[integer - min]++;
    int x = 0;
    for(int count = 0; count<range; count++)
        while (holes[count]-- > 0) {
            list.set(x, count + min);
            x++;
        }
    long endTime   = System.nanoTime();
    return endTime - startTime;
}
```

Hypothesis

Time

Regardless of distribution of elements are in the list, we have check them individually and place according hole. This would mean O(n) time complexity for placing them in to holes. However replacing them back for a acquiring a sorted list would take another chunk of time dependant on the size of the pigeonhole array which is the range of the list. This would bring our final time complexity to O(N+n). Considering smaller range for list we can achieve best case time complexity when range is 0 it would be Θ(n).

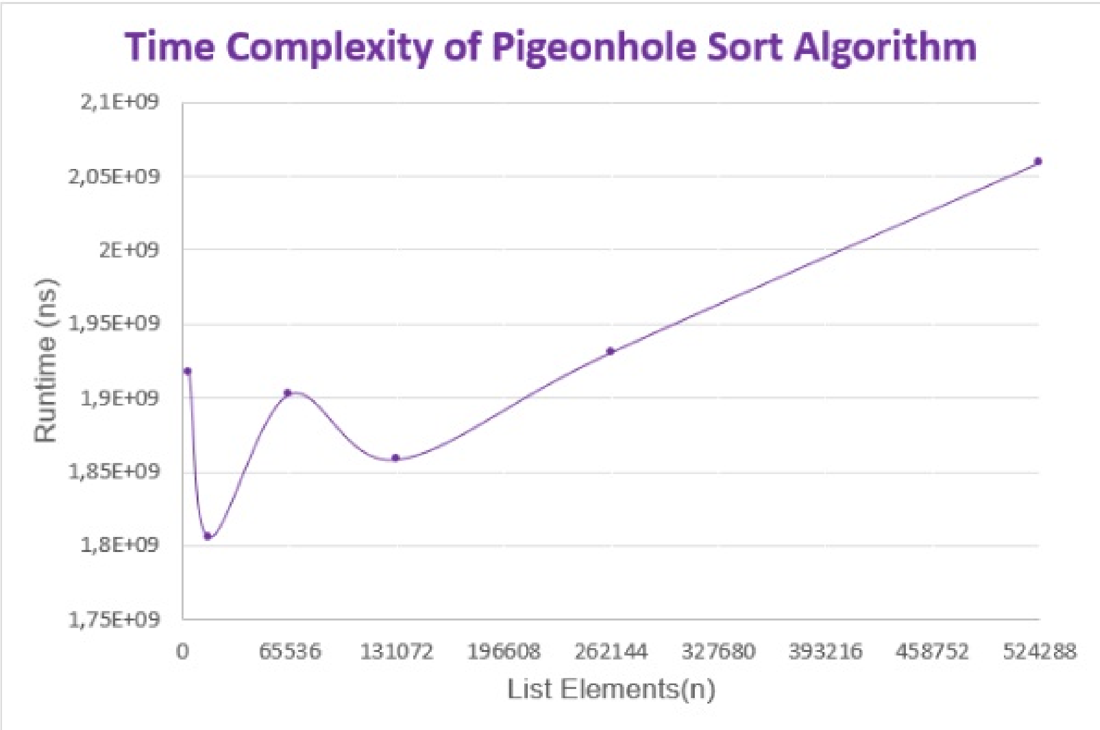
Space

According to range of the given list another pigeonholes list has to be created. This would immediately bring the space complexity up to O(N). However in the further functions of the algorithm we fill that list with all the elements in our list as well and at the final analysis we would need a piece of space according O(N+n) space complexity.

Testing Results

Performed on same randomly generated integers of range 0-1147483647.

Pigeonhole Sort	Test No 1	Test No 2	Test No 3	Test No 4	Test No 5	Avarage
4096	2275425001	1678104400	1837179900	1933440000	1863479200	1917525700
16384	1844626000	1875615801	1788033301	1758911400	1761575500	1805752400
65536	2286213800	1824588300	1820933899	1772447900	1807602300	1902357240
131072	1839962100	1865129901	1909338100	1899487000	1776422700	1858067960
262144	1797878600	1951479400	1996538499	1942606000	1963806300	1930461760
524288	1921757700	2087032301	2184250501	2085135400	2015191100	2058673400



Conclusion

Looking at the runtime data we can clearly see for small count of elements compared to wide range of integers, even with reduced range since my memory couldn't fit an array as big, the range of list is way more deterministic in runtime. This algorithm would be useful for lists with smaller range and takes lots of unnecessary space for widely cast elements.

Bitonic Sort Algorithm

With bitonic sort algorithm we are aiming to achieve bitonic form of the list. This is done by dividing list into smaller sub lists and merging them back.

```java
    static public long bitonic(List<Integer> A){
        Scanner scan = new Scanner(System.in);
        System.out.println("Please state direction of bitonic sorting.");
        int direction = Integer.parseInt(scan.nextLine());
        long startTime = System.nanoTime();
        sorter.bitonicSort(A, A.size(), direction);
        long endTime   = System.nanoTime();
        return endTime - startTime;
    }

    static public void compAndSwap(List<Integer> a, int i, int j, int dire){
        if ( (a.get(i) > a.get(j) && dire == 1) || (a.get(i) < a.get(j) && dire == 0))
        {
            int temp = a.get(i);
            a.set(i,a.get(j));
            a.set(j,temp);
        }
    }

    static public void bitonicMerge(List<Integer> a, int low, int cnt, int dire){
        if (cnt>1)
        {
            int k = cnt/2;
            for (int i=low; i<low+k; i++)
                compAndSwap(a,i, j: i+k, dire);
            bitonicMerge(a,low, k, dire);
            bitonicMerge(a, low: low+k, k, dire);
        }
    }

    static public void bitonicSort(List<Integer> a, int low, int cnt, int dire)
    {
        if (cnt>1)
        {
            int k = cnt/2;

            bitonicSort(a, low, k, dire: 1);
            bitonicSort(a, low: low+k, k, dire: 0);
            bitonicMerge(a, low, cnt, dire);
        }
    }
```

Hypothesis

Time

Regardless of number or distribution of elements are in the list, we have recursively divide and swap all the items in the list. We can perform this operation at parallel time. Resulting sorting networks consist of O(n$\log^2(n)$) comparators and have a delay of $O(\log^2(n))$, where n is the number of items to be sorted. This would bring the time complexity on all cases to $O(\log^2(n))$.

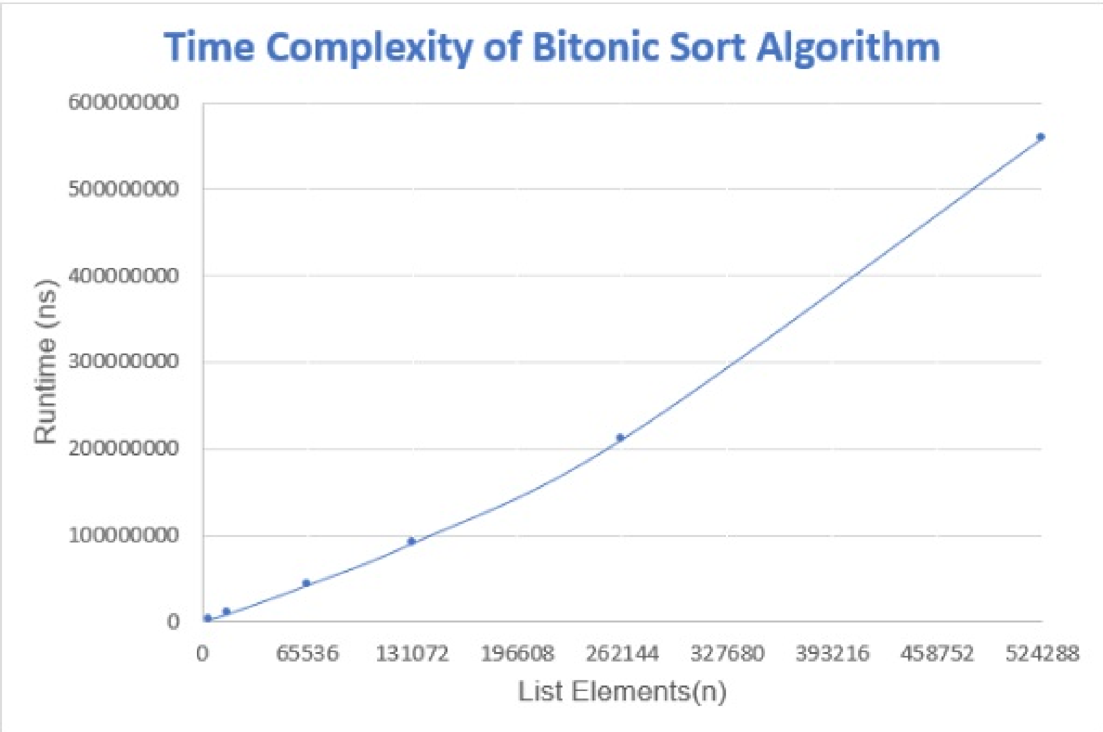
Space

Storing network of divided lists would bring space complexity to O(n log^2n).

Testing Results

Performed on same randomly generated integers of range 0-1147483647.

Bitonic Sort	Test No 1	Test No 2	Test No 3	Test No 4	Test No 5	Avarage
4096	6536400	1865300	1584101	1672500	1612000	2654060,2
16384	14792800	8103901	8085099	8708000	8410500	9620060
65536	45378301	43518601	42135900	40095800	41632600	42552240,4
131072	89549499	90080600	94740300	92044600	88774000	91037799,8
262144	208228601	210712800	220099600	213943200	203607100	211318260,2
524288	942056100	453298601	454234499	482985400	462177800	558950480



Conclusion

At average performing well and chart seems linear. However that is due to not performing on bigger lists.

Comb Sort Algorithm

With the comb sort algorithm we perform swap operation on elements distanced by a shrinking gap until we reach sorted list. Starting with bigger gap points and shrinking gap deals with turtle elements.

```java
static public long comb(List<Integer> A){
    long startTime = System.nanoTime();
    int gap = A.size();
    double shrink = 1.3;
    boolean sorted = false;
    while (!sorted) {
        gap = (int) (gap / shrink);
        if (gap <= 1) {
            gap = 1;
            sorted = true;
        }
        int i = 0;
        while (i + gap < A.size()) {
            if (A.get(i) > A.get(i + gap)) {
                sorter.swapKeys(A , i , j: i + gap );
                sorted = false;
            }
            i++;
        }
    }
    long endTime   = System.nanoTime();
    return endTime - startTime;
}

static public void swapKeys(List<Integer> A, int i, int j){
    int temp;
    temp = A.get(i);
    A.set(i,A.get(j));
    A.set(j,temp);
}
```

Hypothesis

Time

When the list is sorted all needed is to confirm it is sorted. With comb sort algorithm reaching smallest gap (1) and checking the function as if we are checking with cocktail sort algorithm would confirm that. Confirming part normally would take O(n) time but while reaching final gap we perform extra operations for each iteration. This would make best case time complexity Θ(nlog n). For the worst case we need to consider gap sequences and that would make O(n^2) time complexity and as for the average we should consider number of increments done by the gap which equals average case time complexity to $\Omega(n^2/2^p)$.

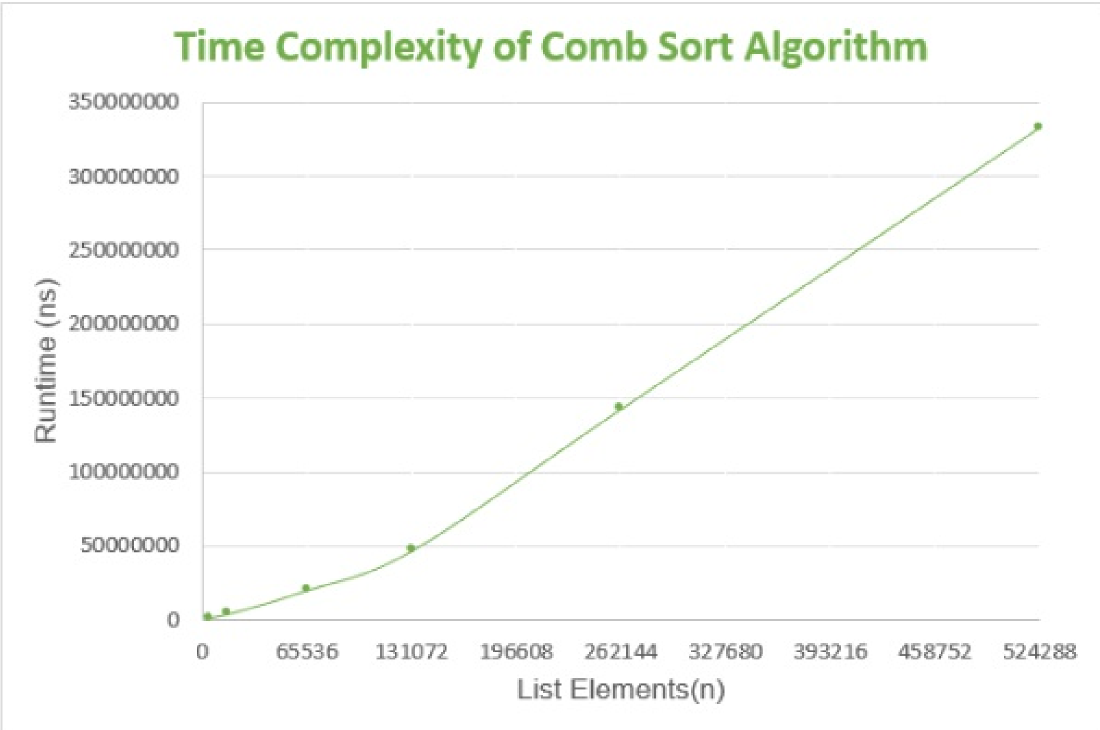
Space

Comb sort algorithm performs swap operations on the given list so it does not allocate any extra memory than temporary memory used for swap operations. Which makes it's space complexity O(1)

Testing Results

Performed on same randomly generated integers of range 0-1147483647.

Comb Sort	Test No 1	Test No 2	Test No 3	Test No 4	Test No 5	Avarage
4096	7168100	730400	722000	725100	793600	2027840
16384	7168700	3924500	4108700	3983400	4093600	4655780
65536	21136900	20562999	20154599	20933500	20036600	20564919,6
131072	48833501	46611700	45284000	45988400	49153500	47174220,2
262144	138472600	141076200	144779800	147121200	143547900	142999540
524288	350890500	348791400	329132801	322595200	313467800	332975540,2



Conclusion

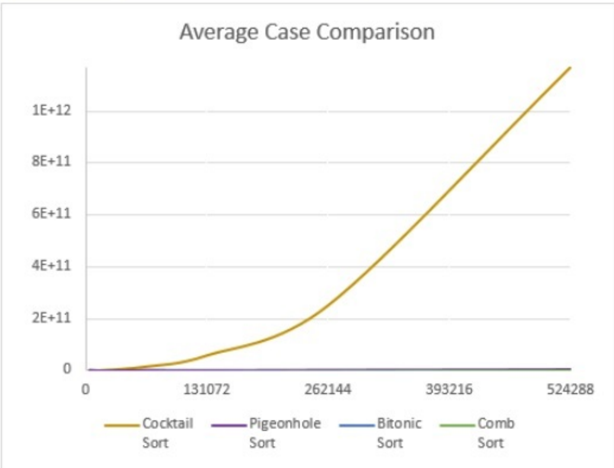
Values hit the smallest so far. There might have been an hardware issue with Test No 1's first two iterations because the result seems to be unexpected. Regardless Comb Sort performs well.

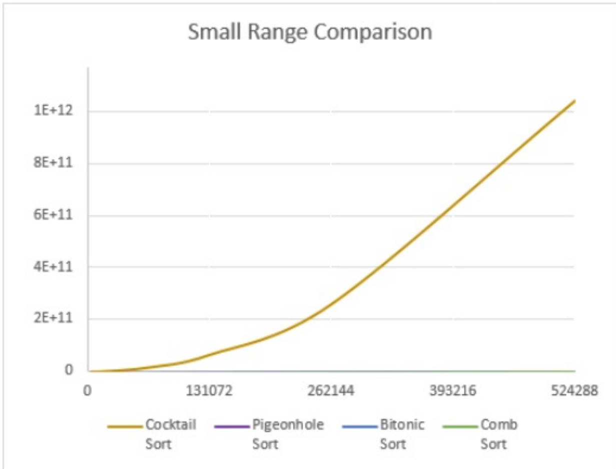
Comparison of Algorithms

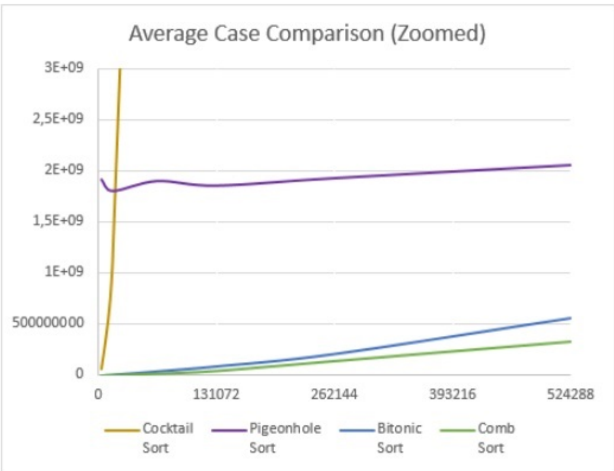
Several Test Results

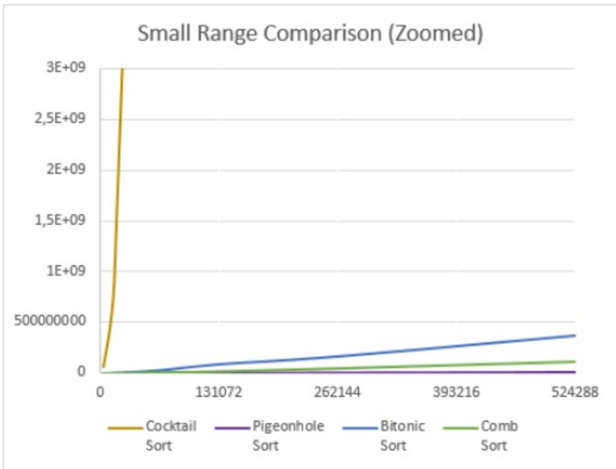
Avarage Case	Cocktail Sort	Pigeonhole Sort	Bitonic Sort	Comb Sort
4096	55555460,4	1917525700	2654060,2	2027840
16384	1100672560	1805752400	9620060	4655780
65536	15494690340	1902357240	42552240,4	20564919,6
131072	58787129100	1858067960	91037799,8	47174220,2
262144	2,52758E+11	1930461760	211318260,2	142999540
524288	1,17407E+12	2058673400	558950480	332975540,2

Small Range	Cocktail Sort	Pigeonhole Sort	Bitonic Sort	Comb Sort
4096	64246300	29300	1092000	403700
16384	955976300	115400	5874900	1894900
65536	16239937300	504200	30519800	9325800
131072	64602069600	979800	88017200	18440500
262144	2,59268E+11	2057700	164389800	45849200
524288	1,04163E+12	3949500	364215400	106332400







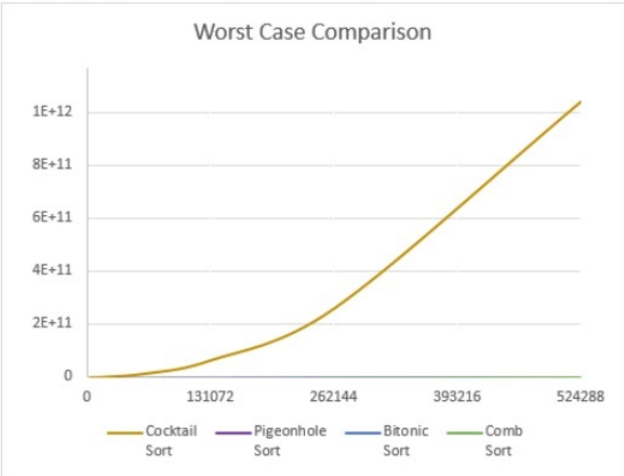


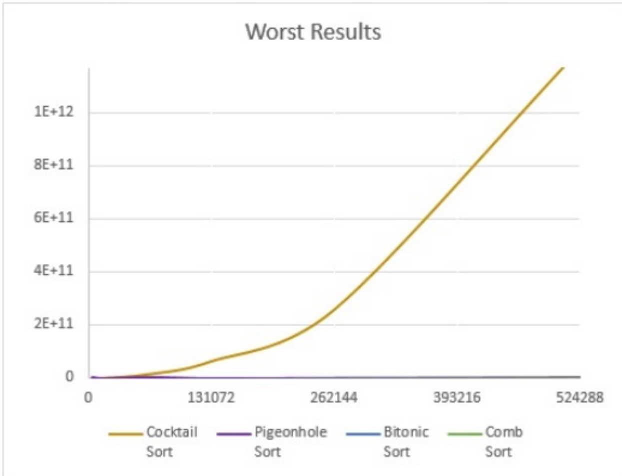
Conclusion

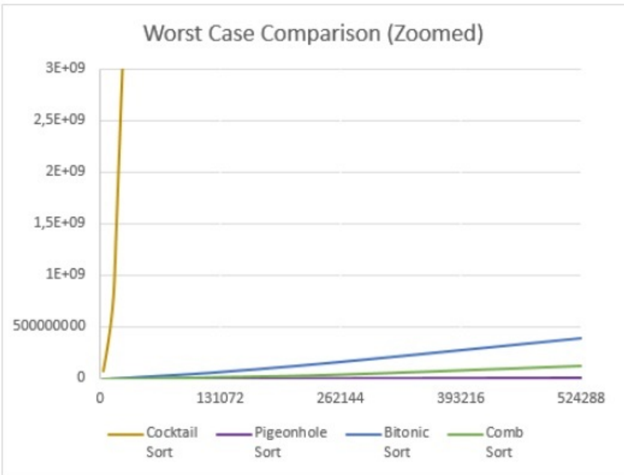
Here we can see at average Cocktail Sort perform significantly terrible compared to others. When range is kept wide pigeonhole sort has worse time result however the slope is still smaller than others. Interestingly when we keep range very small (0-100 in this case) we can clearly see pigeonhole sort outperforms others. Otherwise it would have been comb sort.

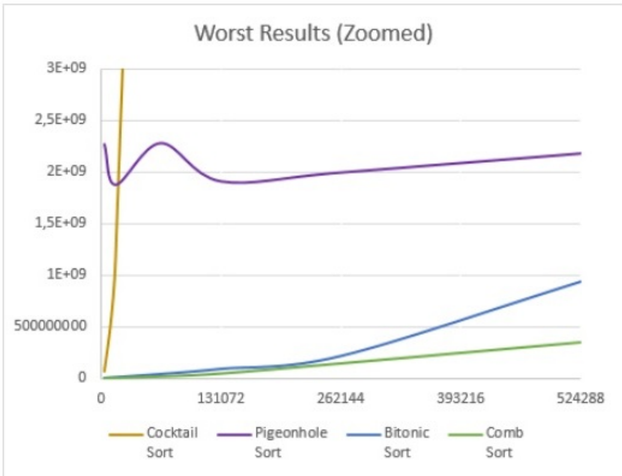
Worst Case	Cocktail Sort	Pigeonhole Sort	Bitonic Sort	Comb Sort
4096	74179400	67000	1193800	369400
16384	984911600	129800	6126100	1828500
65536	16081386400	517300	31541900	8379200
131072	64246731800	973300	67943900	18146600
262144	2,59664E+11	2288800	164173200	42379000
524288	1,04156E+12	4156900	388459600	123205200

Worst Result	Cocktail Sort	Pigeonhole Sort	Bitonic Sort	Comb Sort
4096	74179400	2275425001	6536400	7168100
16384	1184311401	1875615801	14792800	7168700
65536	16277307801	2286213800	45378301	21136900
131072	64602069600	1909338100	94740300	49153500
262144	2,59664E+11	1996538499	220099600	147121200
524288	1,24133E+12	2184250501	942056100	350890500







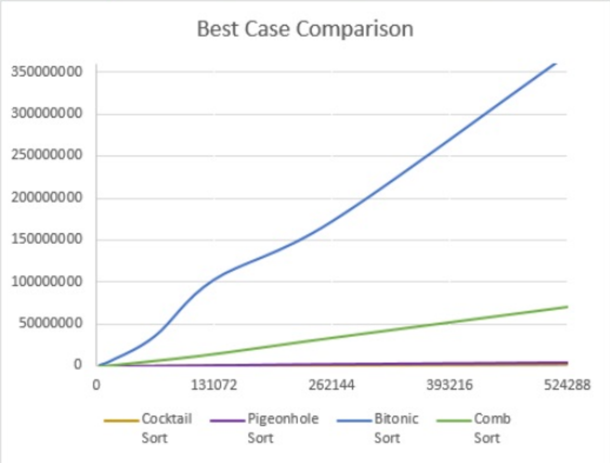


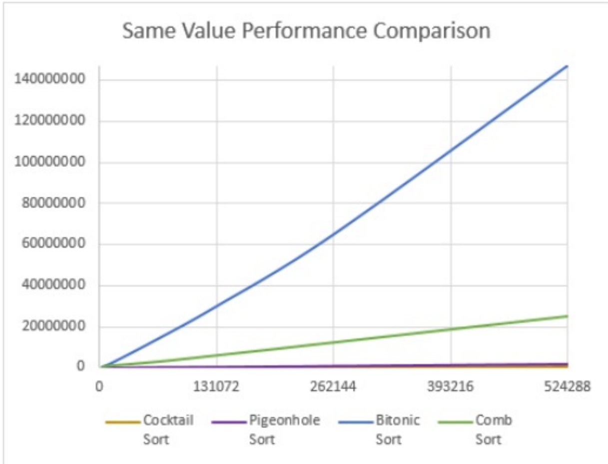
Conclusion

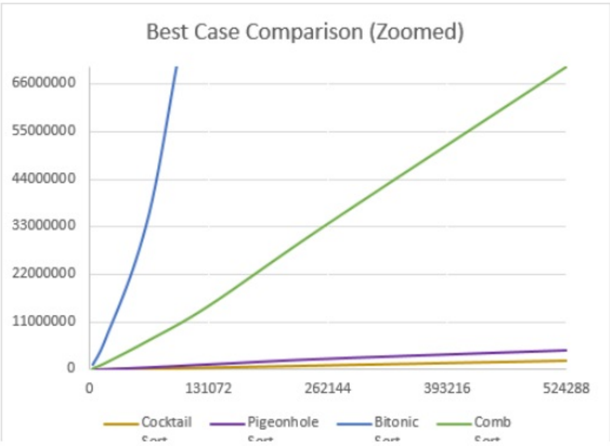
For the Worst Case test I run reversed list. Cocktail Sort seems to not care about worst case and still perform similarly. Pigeonhole sort performed better since the range of the list was equal to size of it. However for pigeonhole sort worst case would mean a bigger range list and it would even go beyond memory capability and crash. So we can not consider pigeonhole performing actually good here. On the right side are worst results picked from all the tests I run and we can clearly see pigeonhole performing worse. However comb sort is performing still good even tho it has worst time complexity of O(n^2). That is because the worst case for comb sort is unique and determined by the gap sequence it has. I have done at least 2 days worth of research on this subject but couldn't find how to generate worst case lists for comb sort algorithm with required sizes.

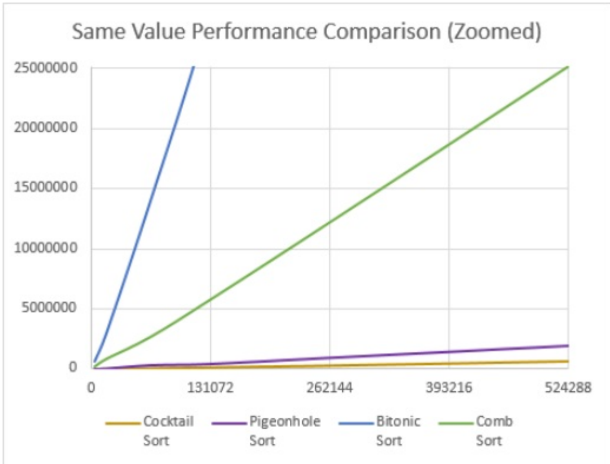
Best Case	Cocktail Sort	Pigeonhole Sort	Bitonic Sort	Comb Sort
4096	16600	43200	1183200	326000
16384	53500	139800	6208200	1427500
65536	223700	565800	35791000	6839800
131072	474200	1238200	103025200	14793800
262144	958400	2527000	172528000	33867000
524288	1962100	4432300	371721700	70280300

Same Values	Cocktail Sort	Pigeonhole Sort	Bitonic Sort	Comb Sort
4096	4800	17200	595800	150000
16384	19300	56300	2815000	760500
65536	69600	328600	14004900	2619100
131072	140700	434400	29856300	5722400
262144	284800	931900	64895800	12161700
524288	629900	1911000	147198600	25165000



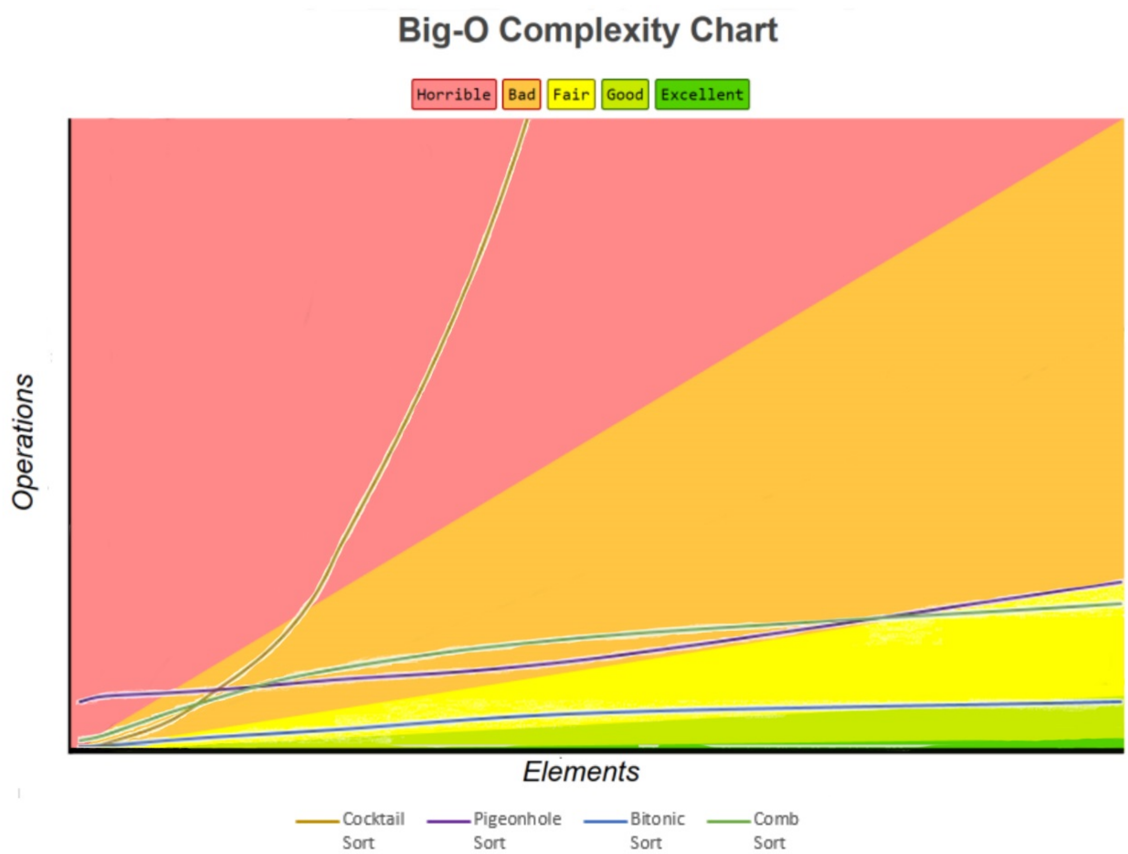






Conclusion

For the best case test I tested already sorted list. As expected the cocktail sort which has had worst performance up until now has the best performance when it comes to best case, because it only takes O(n) time which is theoretically the best possible time complexity. On the other hand bitonic sort seems to not care about whatever the list is sorted or not, while comb sort has better results but similar to reversed sorted list. That is because for comb sort can deal with turtle values efficiently and sort reverse sorted lists as fast as sorted lists. On the right side I tested same values instead of sorted values and results were far more impressive but the hierarchy didn't change.



Concluding Questions

- In which range of n, your best algorithm significantly outperforms the others and why?
  As the n gets larger the best algorithm will outperform the others as logically. In our case we can consider 2 different situations; First is range of list is small and then it would be the pigeonhole sort algorithm because it works at O(N-n) time, second case would be N is large, then it is bitonic sort performing best because it has the smallest average time complexity.

- In which range of n, the algorithm(s) you deal with, get worse and why?
  As n gets bigger cocktail sort algorithm performs significantly worse because of it's asymptotic time complexity. However as N gets larger even tho if the n is small pigeonhole sort would perform worse than cocktail sort.

- Which of the algorithm(s) you deal with behave(s) fine even on worst case scenario?
  For the smaller range N, we can say pigeonhole, bitonic and comb sort performs fine however considering worst case for pigeonhole as N being large only bitonic sort and comb sort pass the worst case test. (This is according to my results; theoretical comb sort has O(n^2) worst time complexity so only bitonic sort would pass the worst case test.)

- Which of the algorithm(s) you deal with behave(s) badly on average and worst case scenarios?
  For the large range of N pigeonhole performs badly, even with n of 2 it would require lots of memory and time. Very bad design in this case. Also cocktail sort perform badly on average reaching hours of runtime.

REFERENCES

Assignment Paper
LaTex Tutorials
Code Project