The goal of this project was to implement and compare the runtimes for 4 different selection algorithms that accept an unsorted array and return the kth smallest value in that array. The 4 algorithms involved sorting with mergeSort then returning the kth value, iterative and recursive QuickSelect that partition the array until the pivot equals the kth value, and recursive QuickSelect using the median of medians as a pivot value.

1. Data Sets, Test Strategies, and Explanation of Results

Table 1-1: Average Algorithm Runtime for Array Size n = 10 to 20000000

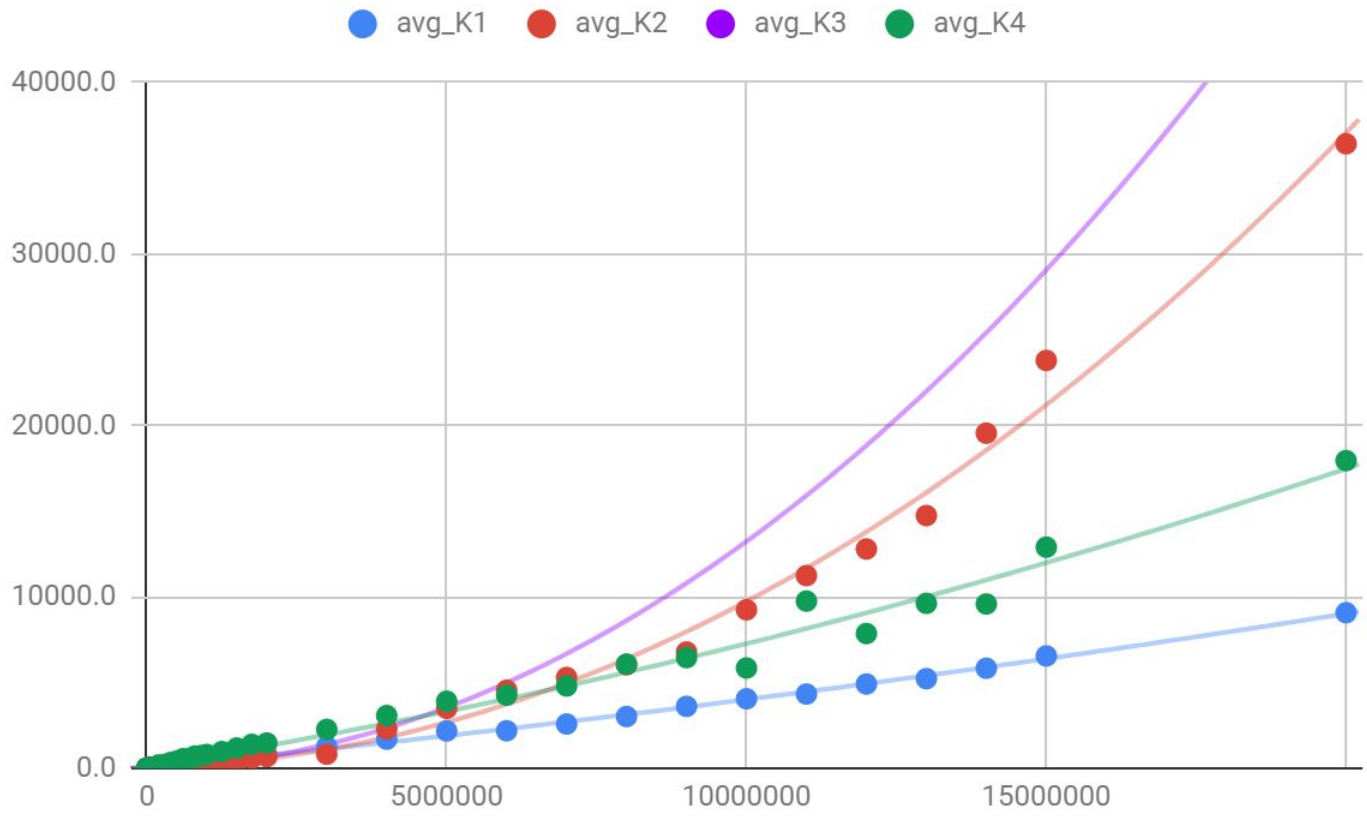
Array Size (n)	Average Algorithm Time (ms)				Important Points
	(K1) MergeSelect	(K2) Iterative QuickSelect	(K3) Recursive QuickSelect	(K4) MedianSelect	
10	0	0	0	0	
50	0	0	0	0.2	
100	0	0	0	0	
250	0.2	0	0	0.8	
500	0.8	0	0	1	
1000	0	0	0.2	3.8	
2500	0.8	0.2	0.2	6.2	
5000	2.2	0.6	0	8.6	
10000	5	1.6	0.8	14.2	Start incrementing by smaller factors for more data points.
20000	10.2	2.2	2	24.8	
30000	13.8	2.6	3.6	36	
40000	17	5	5	39.6	
50000	20.8	6.6	5	49.6	
100000	28.2	7	6	54.6	
200000	89	25.8	25.8	193	
300000	122.2	41.8	39.6	215.6	
400000	170.8	57.6	56.8	335.8	K3 throws StackOverflow error
500000	215.8	66.6		414.2	
600000	260	106.2		552.2	
700000	280.2	125.6		543	
800000	345	115.8		710.2	
900000	408.2	146.8		732.6	
1000000	404	210.2		798.8	
1250000	515.8	303.8		969.6	
1500000	618.8	404.8		1179	
1750000	786.8	578.6		1391.8	
2000000	771.2	676.8		1468.6	
3000000	1257.6	819.2		2265.6	
4000000	1695.8	2289.8		3071	
5000000	2172.8	3492		3904.8	
6000000	2185	4547		4261	
7000000	2577	5291		4797	
8000000	3014	6035		6075	
9000000	3608	6779		6453	
10000000	4052	9241		5843	K1 & K4 are closest in time
11000000	4328	11235		9748	
12000000	4912	12788		7855	
13000000	5225	14736		9618	
14000000	5830	19542		9580	
15000000	6550	23785		12893	Ended testing when array generator threw OutOfMemory error
20000000	9070	36423		17945	

1. Data Sets, Test Strategies, and Explanation of Results (continued)

The table above features data collected by averaging the runtimes for each algorithm, selecting the kth value to be k = 1, n/4, n/2, 3n/4, and n, where n is the size of the array. For each test, the program generates an array of size n populated with random numbers between 1-n*10, and tests all 4 algorithms using the same array. Each algorithm is run 10 times for a single k value and the runtime for all 10 trials is averaged. Then all 5 kth values (1, n/4, n/2, 3/4n, n) are averaged for the final runtimes.

Fig. 1-2: Graph of Array Size vs Algorithm Runtimes

Key: K1 = MergeSelect K2 = Iterative QuickSelect K3 = Recursive QuickSelect K4 = MedianSelect



The expected runtimes were: K1 = O(nlogn), K2/K3 between Ω(n) and O(n^2), K4 = O(n). K4, MedianSelect, was thus expected to be the fastest, because each time the algorithm is picking the best possible value to pivot around and therefore reducing the number of recursive calls and values being compared each time.

However, when examining the results it turns out that K1, the MergeSelect algorithm, had the fastest average runtime out of the 4. This may be due to the simplicity of the algorithm, as the recursive sorting is the most simple way to handle an array of such large size, as it divides the array in half then merges each half until everything is sorted. Unlike selecting a pivot, there are no complex math operations or internal sorting, so the algorithm is the most straightforward out of the 4.

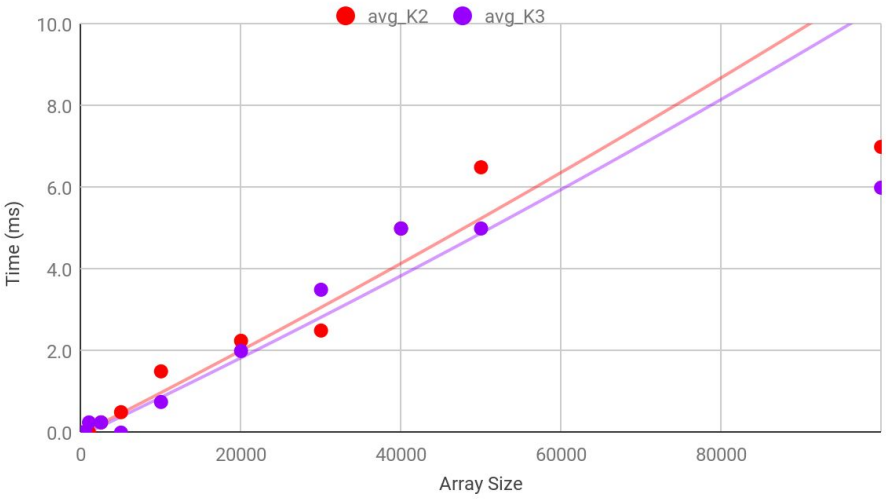
The 2 QuickSelect algorithms performed similarly to one another in polynomial time, although the iterative version had a faster runtime for larger array sizes. As mentioned in Table 1.1, the recursive QuickSelect algorithm threw a stack overflow error halfway through the tests at size n = 400000. This is because due to the random assortment of numbers and the fact that the pivot was chosen to be random each time, there could be up to n^2 recursive calls to find the correct pivot. 400000^2 = 160000000000, which is far beyond the maximum size tested and the likely reason for the error. Both K2 and K3 appear on the graph with a line of best fit matching O(n^2).

K4, the MedianSelect algorithm, outperformed both K2 and K3 as expected, because it selected an optimal value for the pivot each time instead of picking the first value at random. The graph shows that the runtime is in nearly linear time, as expected by the theoretical complexity. However, due to the program terminating with an OutOfMemory error, it was unable to be determined whether K4 could surpass K1 with a faster runtime.

Testing was terminated at size n = 20000000 when the program threw a java OutOfMemory error when trying to build an array of that size.

2. Select 2 vs Select 3

Fig. 2-1: K2 vs K3 Runtime



The above graph is a closeup of the first few data points for K2 and K3, up to size n=100000. This graph shows that initially the recursive QuickSelect (K3) is actually faster (takes less time) than the iterative algorithm. This is because at small array sizes the recursive algorithm takes less time to find the pivot than iterating through each possible pivot in n, as less recursive calls can potentially be made. However, referring back to Fig 1-2, the projected growth for the recursive algorithm far exceeds the actual growth of the iterative algorithm, meaning that for large array sizes the iterative algorithm is optimal due to the lack of recursion. Unfortunately the excessive recursive calls led to a StackOverflow error, so it was unable to be verified if the recursive algorithm actually grew at the projected rate shown by the best fit line.

3. Select 4 vs Select 1

Theoretically, there should be a point where the MergeSelect algorithm (K1) becomes slower than the MedianSelect algorithm (K4). This is because K1 has a theoretical complexity of O(nlogn) due to the steps it takes to first sort the array. K4 has a theoretical worst case of O(n) because the value chosen as the pivot will always be the median of median values, that is, it should always be the absolute median in the array, which divides the data set evenly each time to allow for faster partitioning to find the kth value. O(n) < O(nlogn), so it was expected that K4 is the fastest algorithm. In practice, however, it can be seen in Fig. 1-2 that K4 never becomes faster than K1. The gap between them started to decrease for a time, but the curves and data between the two never crossed. The last point at which the two were closest was size n = 10000000, after which the difference between the two data sets continued to increase again, widening the runtime difference between them.

4. Strength and Constraints

The MedianSelect algorithm utilized QuickSelect partitioning with a best case of Ω(n), and guaranteed that best case through the use of the median of medians value as a pivot. Therefore, in theory this algorithm should have given the best runtime when compared to the O(nlogn) runtime of MergeSelect and potential $O(n^2)$ runtime of the 2 QuickSelect algorithms. However, in practice this algorithm was less effective than MergeSelect, at least for this implementation.

The strength of the implementation of each algorithm was in the simplicity of the MergeSelect and 2 QuickSelect algorithms. All 3 of the aforementioned methods are easy to read and understand, as they follow the pseudocode for the algorithms closely. By contrast, however, that simplicity in implementation did not carry over to the MedianSelect algorithm, which involves less optimized code due to a lack of understanding in the difference between the median of medians being used as a pivot rather than a value to be calculated separately then used as a pivot. It is likely that with more manipulation of the array indices the MedianSelect algorithm could have been completed more concisely with fewer methods.

5. Program Correctness: Java Code

Selection Algorithm class

```
import java.util.*;
class Selection
{
public int mergeSelect(int[] arr, int k)
{
          mergeSort(arr, 0, arr.length-1);
          return arr[k-1];
}

private void mergeSort(int[] arr, int left, int right)
{
          if (left<right)     {
                    int mid = (left+right)/2;
                    mergeSort(arr,left,mid);
                    mergeSort(arr,mid+1,right);
                    merge(arr,left,mid,right);
          }
}

private void merge(int[] arr,int left,int mid,int right)
{
                    int Lsz = mid-left+1;
                    int Rsz = right-mid;
                    int[] Larr = new int[Lsz];
                    int[] Rarr = new int[Rsz];
                    for(int i=0;i<Lsz;i++)
                              Larr[i] = arr[left+i];
                    for(int i=0;i<Rsz;i++)
                              Rarr[i] = arr[mid+i+1];

                    int l=0, r=0, k=left;
                    while(l<Lsz && r<Rsz)         {
                              if(Larr[l]<=Rarr[r])
                                        arr[k] = Larr[l++];
                              else
                                        arr[k] = Rarr[r++];
                              k++;
                    }

                    while(l<Lsz && k<arr.length)
                              arr[k++] = Larr[l++];

                    while(r<Rsz && k<arr.length)
                              arr[k++] = Rarr[r++];
}

public int quickIterSelect(int[] arr, int k, int left, int right)
{
          int pivot = partition(arr, left, right, arr[left]);
          while(pivot!=(k-1)) {
                    if(pivot>(k-1))
                              right=pivot-1;
                    else
                              left=pivot+1;
                    pivot = partition(arr, left, right, arr[left]);
          }
          return arr[pivot];
}

//partitions around pivot element
private int partition(int arr[], int left, int right, int pivot)
{
          int i=left;
          while(arr[i]!=pivot)
                    i++;
          swap(arr, i, right); //swap last value w/ pivot

          i=left;
          for(int x=left;x<=right-1;x++){
                    if(arr[x]<=pivot)   {
                              swap(arr,i,x);
                              i++;
                    }
          }
          swap(arr,i,right);
          return i;
}

private void swap(int[] arr, int idx1, int idx2)
{
          int temp = arr[idx1];
          arr[idx1] = arr[idx2];
          arr[idx2] = temp;
}
```

```
public int quickRecSelect(int[] arr, int k, int low, int high)
{
          int pivot = partition(arr, low, high, arr[low]);
          if(pivot==(k-1))
                    return arr[pivot];
          else if(pivot>(k-1))
                    return quickRecSelect(arr, k, low, pivot-1);
          else
                    return quickRecSelect(arr, k, pivot+1,high);
}

public int medianSelect(int[] arr, int k, int left, int right)
{ //call selection on k-1, bc methods use array idx 0
          return arr[select(arr,left,right,k-1)];
}

private int select(int[] arr, int left, int right, int k)
{
          while(left<=right)  {
                    if(left==right)
                              return left; //return left
                    int pivot = pivot(arr, left, right);
                    pivot = partition(arr, left, right, pivot, k);
                    if(pivot==k)
                              return k; //return kth index
                    else if(pivot > k)
                              right = pivot-1;
                    else
                              left = pivot+1;
          }
          return -1; //not found
}

private int pivot(int[] arr, int left, int right)
{
          if(right-left<5) //when <=5 get median directly
                    return median5(arr,left,right);
          int median = 0, subright;
          for(int i=left;i<right;i+=5)  {
                    //get median position
                    subright = i+4;
                    if(subright>right)
                              subright=right;
                    median = median5(arr,i,subright);
                    int pos = left + (i-left)/5;
                    //swap median & position
                    swap(arr,median,pos);
          }
          int mid = right-left / (10+left+1);
          subright = left+(right-left)/5;  //reuse var from loop
          return select(arr,left,subright,mid);
}

private int partition(int[] arr,int left,int right,int pivot,int k)
{
          int value = arr[pivot];
          arr[pivot] = arr[right];
          arr[right] = value;
          int index = left;
          for(int i=left;i<right-1;i++) {
                    if(arr[i] < value)  {
                              swap(arr,i,index);
                              index++;
                    }
          }
          int indexEq = index;
          for(int j=index;j<right-1;j++){
                    if(arr[j] == value) {
                              swap(arr,j,indexEq);
                              indexEq++;
                    }
          }
          swap(arr,right,indexEq);
          if(k<index)
                    return index;
          else if(k<=indexEq)
                    return k;
          else
                    return indexEq;
}

private int median5(int[] arr, int left, int right)
{//called only on arrays of <=5 elements
          mergeSort(arr,left,right);
          return (left+right)/2;
}
} //end class Selection
```

Test Methods class

```
import java.util.*;
class Test
{
private Selection select;
private int[] array;

public Test()
{
          select = new Selection();
}

public void buildArray(int len)
{
          array = new int[len];
          int i=0;
          while(i<len) {
                    array[i] = (int)(Math.random()*(len*10)+1);
                    i++;
          }
}

public long timeCounter(int k, int kselect)
{
  long startTime, runTime, avg = 0;
  int value = 0;
  for(int i=0;i<1;i++)
  {
          startTime = System.currentTimeMillis();
          if(kselect==1)
          {
          value = select.mergeSelect(array,k);
          }
          else if(kselect==2)
          {
          value = select.quickIterSelect(array,k,0,array.length-1);
          }
          else if(kselect==3)
          {
          value = select.quickRecSelect(array,k,0,array.length-1);
          }
          else
          {
          value = select.medianSelect(array,k,0,array.length-1);
          }
          runTime = System.currentTimeMillis() - startTime;
          avg+=runTime;
  }
          avg/=1;
  System.out.println("\t\t\t" + k + "-th value: " + value +
                    "\t\tRUNTIME: " + avg);
  return avg;
}

private void printArray()
{
          System.out.print("\nTest array: ");
          for(int i:array)
                    System.out.print(i+" ");
          System.out.println();
}

public void testSelection(int n, int k)
{
          System.out.println("\n\nstart test for: n="+ n + " k="+ k);
          //printArray();
          System.out.print("MergeSelect testing...");
          timeCounter(k,1);
          System.out.print("Iterative QuickSelect testing...");
          timeCounter(k,2);
          System.out.print("Recursive QuickSelect testing...");
          timeCounter(k,3);
          System.out.print("MedianSelect testing...");
          timeCounter(k,4);
          //System.out.print("\nend test for: n="+ n + " k="+ k);
}

public void testRuntime(int n)
{
          long k1avg=0, k2avg=0, k3avg=0, k4avg=0;
          buildArray(n);
          System.out.println("\nArraySize: "+n);
          k1avg+=timeCounter(1,1);
          k1avg+=timeCounter(n/4,1);
          k1avg+=timeCounter(n/2,1);
          k1avg+=timeCounter((int)((.75)*n),1);
          k1avg+=timeCounter(n,1);
          k1avg/=5;
          System.out.println("K1 RUNTIME: "+k1avg);
          k2avg+=timeCounter(1,1);
          k2avg+=timeCounter(n/4,1);
          k2avg+=timeCounter(n/2,1);
          k2avg+=timeCounter((int)((.75)*n),1);
          k2avg+=timeCounter(n,1);
          k2avg/=5;
          System.out.println("K2 RUNTIME: "+k2avg);
          k3avg+=timeCounter(1,1);
          k3avg+=timeCounter(n/4,1);
          k3avg+=timeCounter(n/2,1);
          k3avg+=timeCounter((int)((.75)*n),1);
          k3avg+=timeCounter(n,1);
          k3avg/=5;
          System.out.println("K3 RUNTIME: "+k3avg);
          k4avg+=timeCounter(1,4);
          k4avg+=timeCounter(n/4,4);
          k4avg+=timeCounter(n/2,4);
          k4avg+=timeCounter((int)((.75)*n),4);
          k4avg+=timeCounter(n,4);
          k4avg/=5;
          System.out.println("K4 RUNTIME: "+k4avg);
}

public void testKth(int n)
{
          buildArray(n);
          testSelection(n, 1);
          System.out.print("\n_____________");
          testSelection(n, n/4);
          System.out.print("\n_____________");
          testSelection(n, n/2);
          System.out.print("\n_____________");
          testSelection(n, (int)((.75)*n));
          System.out.print("\n_____________");
          testSelection(n, n);
          System.out.print("\n_____________");
}
} //end class Test
```

6. Program Correctness: Output

Below is a test output for an array of size 10, to show that the correct kth value is chosen with each algorithm. Note that for such a small array the runtime is smaller than 1 millisecond, and therefore shows as 0. Numbers generated are between range of 1 to n*10 = 1 - 100

```
Test array: 53 57 54 19 24 87 12 16 42 55
start test for: n=10 k=1
MergeSelect testing...            1-th value: 12           RUNTIME: 0
Iterative QuickSelect testing...  1-th value: 12           RUNTIME: 0
Recursive QuickSelect testing...  1-th value: 12           RUNTIME: 0
MedianSelect testing...           1-th value: 12           RUNTIME: 0
---------------
start test for: n=10 k=2
MergeSelect testing...            2-th value: 16           RUNTIME: 0
Iterative QuickSelect testing...  2-th value: 16           RUNTIME: 0
Recursive QuickSelect testing...  2-th value: 16           RUNTIME: 0
MedianSelect testing...           2-th value: 16           RUNTIME: 0
---------------
start test for: n=10 k=5
MergeSelect testing...            5-th value: 42           RUNTIME: 0
Iterative QuickSelect testing...  5-th value: 42           RUNTIME: 0
Recursive QuickSelect testing...  5-th value: 42           RUNTIME: 0
MedianSelect testing...           5-th value: 42           RUNTIME: 0
---------------
start test for: n=10 k=7
MergeSelect testing...            7-th value: 54           RUNTIME: 0
Iterative QuickSelect testing...  7-th value: 54           RUNTIME: 0
Recursive QuickSelect testing...  7-th value: 54           RUNTIME: 0
MedianSelect testing...           7-th value: 54           RUNTIME: 0
---------------
start test for: n=10 k=10
MergeSelect testing...            10-th value: 87          RUNTIME: 0
Iterative QuickSelect testing...  10-th value: 87          RUNTIME: 0
Recursive QuickSelect testing...  10-th value: 87          RUNTIME: 0
MedianSelect testing...           10-th value: 87          RUNTIME: 0
---------------
Process finished with exit code 0
```

Below is an example output for a larger array to show runtimes. The array is not printed due to its large size. Range is between 1-400000*10 = (1-4000000)

```
start test for: n=400000 k=1
MergeSelect testing...            1-th value: 1            RUNTIME: 251
Iterative QuickSelect testing...  1-th value: 1            RUNTIME: 36
Recursive QuickSelect testing...  1-th value: 1            RUNTIME: 32
MedianSelect testing...           1-th value: 1            RUNTIME: 298
---------------
start test for: n=400000 k=100000
MergeSelect testing...            100000-th value: 315021  RUNTIME: 153
Iterative QuickSelect testing...  100000-th value: 315021  RUNTIME: 60
Recursive QuickSelect testing...  100000-th value: 315021  RUNTIME: 67
MedianSelect testing...           100000-th value: 315021  RUNTIME: 422
---------------
start test for: n=400000 k=200000
MergeSelect testing...            200000-th value: 1929714 RUNTIME: 161
Iterative QuickSelect testing...  200000-th value: 1929714 RUNTIME: 77
Recursive QuickSelect testing...  200000-th value: 1929714 RUNTIME: 76
MedianSelect testing...           200000-th value: 1929714 RUNTIME: 575
---------------
start test for: n=400000 k=300000
MergeSelect testing...            300000-th value: 2167360 RUNTIME: 160
Iterative QuickSelect testing...  300000-th value: 2167360 RUNTIME: 61
Recursive QuickSelect testing...  300000-th value: 2167360 RUNTIME: 67
MedianSelect testing...           300000-th value: 2167360 RUNTIME: 481
---------------
start test for: n=400000 k=400000
MergeSelect testing...            400000-th value: 3831004 RUNTIME: 155
Iterative QuickSelect testing...  400000-th value: 3831004 RUNTIME: 56
Recursive QuickSelect testing...  400000-th value: 3831004 RUNTIME: 69
MedianSelect testing...           400000-th value: 3831004 RUNTIME: 69
---------------
Process finished with exit code 0
```