Sorting Algorithms: Review of n^2 and $n * \lg(n)$ Algorithms, and Quicksort with Median of n on an Almost-Sorted Array

Zeph Turner

3/2018

1 Introduction

I will discuss ways to optimize quicksort's performance on a nearly-sorted list using different partitioning strategies and compare the effectiveness of these strategies experimentally.

1.1 Experimental Runtime

Experimental runtimes for all algorithms were tested using Java code compiled with the Eclipse compiler, version 20140604-1726. All algorithms were run on my laptop, which has a 4th-generation Intel Core i7 processor, a clock rate of 3.4 GHz, and 4 cores. Times under 1,000 ms were repeated three times and the average taken, rounded to the nearest tenth. Sorts were performed on an array of doubles of length n randomly generated using `Math.random()`.

2 Quicksort

Quicksort is a sorting algorithm that selects a "pivot" value, partitions the array into two smaller arrays where one contains all values less than the pivot value and one contains all values greater than the pivot value, and then recursively partitions the two smaller arrays. Unlike mergesort, quicksort has no "combine" step; all of the work is done during the call to Partition, the "divide" step.

2.1 Theoretical Runtime

Quicksort has two functions: the recursive function Quicksort, shown in Table 1, and Partition, shown in Table 2.

#	**Pseudocode:** Quicksort(A, p, r)
1	if $p < r$
2	$\quad q = \text{Partition}(A, p, r)$
3	$\quad \text{Quicksort}(A, p, q-1)$
4	$\quad \text{Quicksort}(A, q+1, r)$

Table 1: Quicksort(A, p, r) Pseudocode

.

Every line in Quicksort runs once, but it also creates two sub-problems whose size depends on q, the pivot, and it calls Partition.

Times run	#	**Pseudocode:** Partition(A, p, q, r)
1	1	$x = A[r]$
1	2	$i = p - 1$
$r - p$	3	for $j = p$ to $r - 1$
$r - p - 1$	4	$\quad$ if$(A[j] \leq x)$
$\sum_{i,j} t_{i,j}$	5	$\quad\quad i++$
$\sum_{i,j} t_{i,j}$	6	$\quad\quad$ swap $A[i] \leftrightarrow A[j]$
1	7	swap $A[i+1] \leftrightarrow A[r]$
1	8	return $i + 1$

Table 2: Partition(A, p, q, r) Pseudocode

The variable lines in Partition (5 and 6) run $\sum_{i,j} t_{i,j}$ times, which cannot exceed $r - p - 2$ in any case, so the most frequently run line is line 4. $r - p - 1$ is the problem size, n, as we are partitioning a subset of a matrix from indices p to r. Therefore Partition(A, p, q, r) is $\Theta(n)$.

The runtime of quicksort overall will depend on the order of the input array. In the best case, q is the median (or almost the median, for an even number of items) of $A[p : r]$ in every call of Partition. Then the overall runtime of quicksort is

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

which is the same as mergesort: $\Theta(n * \lg(n))$.

In the worst case, $q = r$ or $q = p$ every time, for an already sorted (ascending or descending) array. In that case the runtime is

$$T(n) = T(n-1) + \Theta(n)$$

The depth of the recursion tree in this case will be n, and there will be $\Theta(n)$ work per level, making this case $\Theta(n^2)$.

The average case might be described by

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + \Theta(n) \quad .$$

The maximum depth of the call tree in this case is $\log_{\frac{3}{2}}(n)$, which is $\Theta(\lg(n))$ (they differ by a constant). The work on each level is still a constant multiple of n. Therefore this case is still $\Theta(n * \lg(n))$.

To conclude, in the best case, quicksort will perform on the order of $n * \lg(n)$, but could perform as badly as n^2 in the worst case. Overall we can only say that quicksort is $\Omega(n * \lg(n))$ and $O(n^2)$. Adaptations that can improve quicksort's worst-case runtime are discussed in Section 4.

2.2 Experimental Runtime

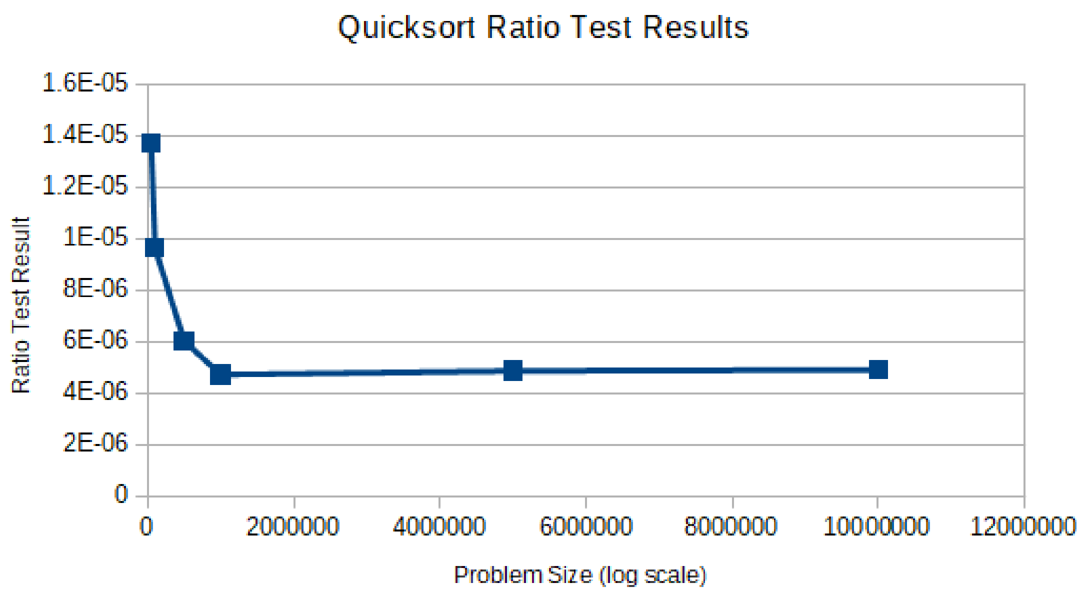
Table 3 shows experimental runtimes for quicksort.

Problem size (n)	**Time (ms)**	**Ratio: Time/$n * \lg(n)$**
50,000	10.7	1.37×10^{-5}
100,000	16	9.63×10^{-6}
500,000	57	6.02×10^{-6}
1,000,000	94	4.71×10^{-6}
5,000,000	541	4.86×10^{-6}
10,000,000	1,142	4.91×10^{-6}

Table 3: Runtimes by Problem Size for Quicksort

Quicksort is experimentally shown to perform on the order of $n * \lg(n)$. The ratios converge to about 5×10^{-6}. The convergence is shown graphically in Figure 1.

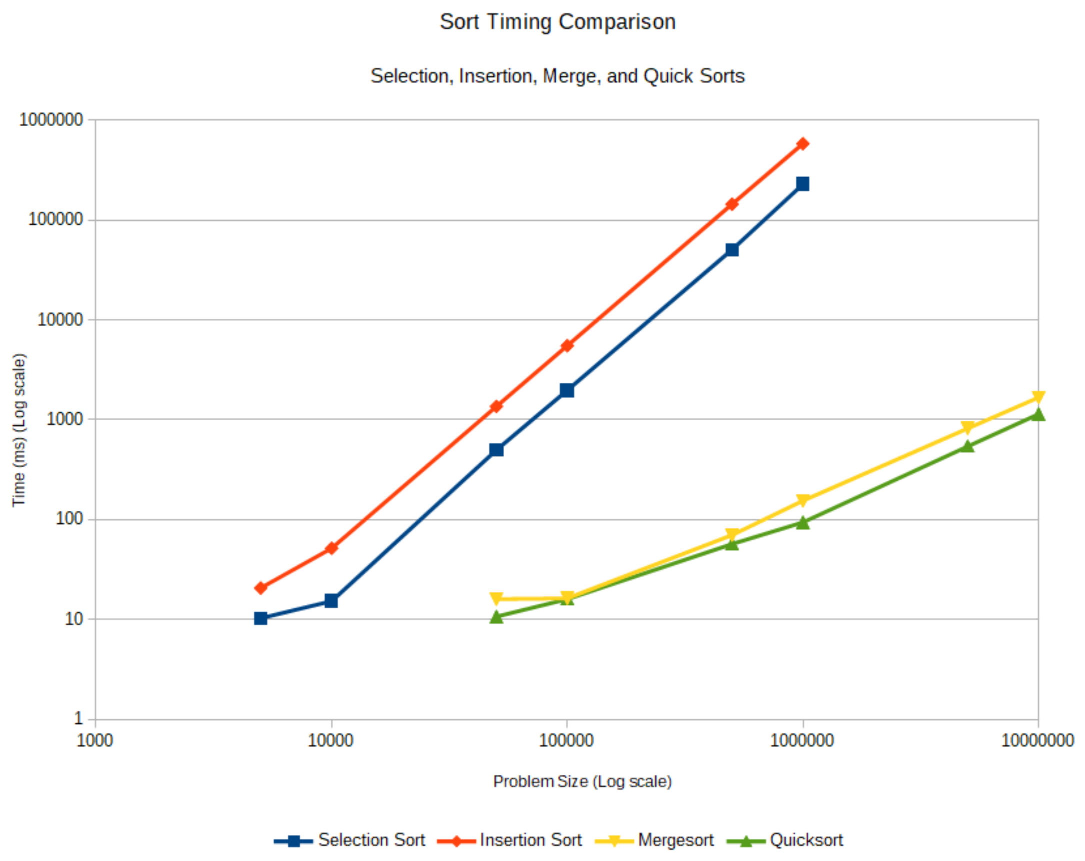
Figure 1: Quicksort Ratio Test



3 Timing Comparison

Figure 2 compares sort times for different problem sizes for quicksort, mergesort, selection sort, and insertion sort.

Figure 2: Comparison of n^2 and $n * \lg(n)$ Sorts



4 Quicksort with Median of n

I investigated an algorithm I call Quicksort with Median of n. Although quicksort was the best-performing sort out of the four I investigated when given randomly arranged lists of numbers, nearly-sorted arrays cause quicksort to degenerate to its worst-case performance by causing bad partitions, where almost all values are less than or greater than the pivot. In the worst case, instead of the ideal recursion $T(n) = 2T\left(\frac{n}{2}\right) + n$, this will result in a recursion more like $T(n) = T(n-1) + n$, which gives performance on the order of n^2. It also creates a very deep call stack because there is a new call to Quicksort for almost every element in the array. On even relatively small problem sizes, it may throw a stack overflow error and be unable to complete the sort at all.

There are several adaptations of quicksort that use a little more computational power to choose pivots more effectively. I looked at two such strategies: taking a pivot from a random index within the subarray passed to Partition, and choosing some number of candidate pivots, finding the median value, and using that as the pivot. This latter strategy is usually implemented as "quicksort with median of 3". It takes 3 values from the array (usually $A[p]$, $A[\lceil 0.5(p+r)\rceil]$, and $A[r]$), finds their median, and uses that value as the pivot in order to avoid bad partitions where the pivot is the maximum or minimum value in the subarray.

I was curious how this algorithm would perform if it took the median of 5, 7, 9, or more values instead. I wrote Quicksort with Median of n to answer this question. Instead of partitioning around $A[r]$, a random index in the subarray, or the median of 3 sampled values, it samples n equally-spaced values from the array and partitions around their median.

To test these algorithms, I generated almost sorted arrays A of size N using the following algorithm, where Math.random() returns a random floating-point number between 0 and 1:[1]

#	**Pseudocode:** Populate A With Almost-Sorted Integers
1	random1 = 12
2	random2 = 2
3	$A[0]$ = (Math.random()*10) + 1
4	for i from 1 to N
5	$A[i] = A[i-1]$ + (Math.random*random1) - random2

Table 4: Generating Almost-Sorted Lists

Usually $A[i] > A[i-1]$, but there is a small chance (about .17) that $A[i] < A[i-1]$. As an example, the following numbers are an array of length 20 generated using this rule, truncated to one decimal place: 8.8, 11.4, 12.7, 19.4, 22.9, 22.3, 25.0, 29.9, 39.4, 46.1, 44.8, 51.8, 52.1, 52.7, 59.8, 66.2, 68.3, 76.0, 82.0. Only one value is out of order; the array is almost-sorted. I sorted these almost-sorted arrays using several different quicksort variants and compared the results.

Note that in this section, n indicates the number of items in Quicksort with Median of n that are considered for the pivot, whereas N indicates the problem size, the length of the full array to be sorted.

4.1 Quicksort with Median of n Pseudocode

I adapted the Partition algorithm from above for quicksort with median of n. The new pseudocode is shown in Table 5.

[1]This algorithm is from Jojonete on this Stack Overflow question: https://stackoverflow.com/questions/22827673/how-to-create-almost-sorted-array-of-100-with-integers-from-1-1000 .

```
#   Pseudocode: PartitionWithMedian($A, p, q, r$)
1   medInd = getMedian($p, r, n$)
2   $x = A$[medInd]
3   swap $A[r] \leftrightarrow A$[medInd]
4   $i = p - 1$
5   for $j = p$ to $r - 1$
6       if($A[j] \leq x$)
7           $i++$
8           swap $A[i] \leftrightarrow A[j]$
9   swap $A[i+1] \leftrightarrow A[r]$
10  return $i + 1$
```

Table 5: PartitionWithMedian(A, p, q, r) Pseudocode

getMedian, shown in Table 6, is a short method that chooses n indices from the array, stores them in a smaller array medianIndices, sorts medianIndices based on the value *at* those indices in the outer array being sorted (A), and returns the median index. If the values in medianIndices are (1, 3, 5) and the values in $A[1], A[3], A[5]$ are (2, 8, 4), at the end of this method medianIndices will be sorted (1, 5, 3): medianIndices itself is not sorted, but A[medianIndices[i]] $\leq A$[medianIndices[$i+1$]].

```
#   Pseudocode: getMedian($A, p, r, n$)
1   if $p < r$
2       return $p$
3   for $i = 0$ to $n - 1$
4       medianIndices[$i$] $= p + i * \frac{r-p}{n-1}$
5   sort medianIndices by values in $A$
6   return medianIndices[$\lfloor n/2 \rfloor$]
```

Table 6: getMedian(A, p, r, n) Pseudocode

getMedian calls a sort function to find the middle value out of the values at the indices in medianIndices. I wrote two variants of quicksort with median of n that use different sorts for this method. One uses insertion sort, which was described in section **??**. The other uses quicksort *without* median of n, which uses the last value in the array in the pivot, which was described in section 2.

Full Java code for both variants is available in the code appendix, section 5, for reference.

4.2 Quicksort with Random Pivot

The last variant of quicksort I tested was quicksort with a random pivot, where the index of the pivot is a randomly generated integer from p to r.

4.3 Timing Comparisons

I compared these algorithms with a variety of n and problem sizes. The results are reported in Table 7. I will abbreviate base quicksort as BQS, random quicksort as RQS, quicksort with median of n and insertion sort to find the median as IQS, n, and quicksort with median of n and base quicksort to find the median as QSQS, n. SO in the table indicates that the sort terminated with a stack overflow. The call stack was too deep to finish the sort.

I tested each sort on increasing problem sizes until either it hit a stack overflow or runtime exceeded 1,000 milliseconds. If the first sort runtime was under 100 milliseconds, I took the average of three times for that problem size.

N	**BQS**	**RQS**	**IQS, 3**	**IQS, 5**	**IQS,** $0.001N$	**QSQS, 3**	**QSQS, 5**	**QSQS,** $0.001N$
5,000	21	5.3	8.3	5	2	2	6	9.3
10,000	47	0.7	9.3	9.6	4.7	4.3	4.3	5.3
20,000	156	8.3	7.7	8	9.3	11.3	6.7	15.3
50,000	SO	10.3	13.3	16	30.3	18	20.7	84.3
100,000		14.3	23.7	22.7	48	22	27.7	537
200,000		17	24.3	37.7	109	31	35	4168
500,000		46.3	46.7	54	498	36	58	
1,000,000		68	75	74.3	1744	64.3	80.7	
2,000,000		125	140	130		113	125	
5,000,000		314	292	319		258	310	
10,000,000		624	622	640		530	628	
20,000,000		1332	1392	1337		987	1182	

Table 7: Timing Comparison for Quicksort Variants

As expected, base quicksort terminated with a stack overflow at a problem size of only 50,000 (recall that I showed earlier that it could sort a *random* array of 10,000,000 values in just over one second!).

Among the quicksort with median of n algorithms, quicksort with median of 3 using base quicksort to find the median performed the best. Among all the algorithms, quicksort with random pivot performed best on smaller problems (N = 5,000 to 500,000) and quicksort with median of 3 using base quicksort performed best on larger problems ($N \geq 1,000,000$).

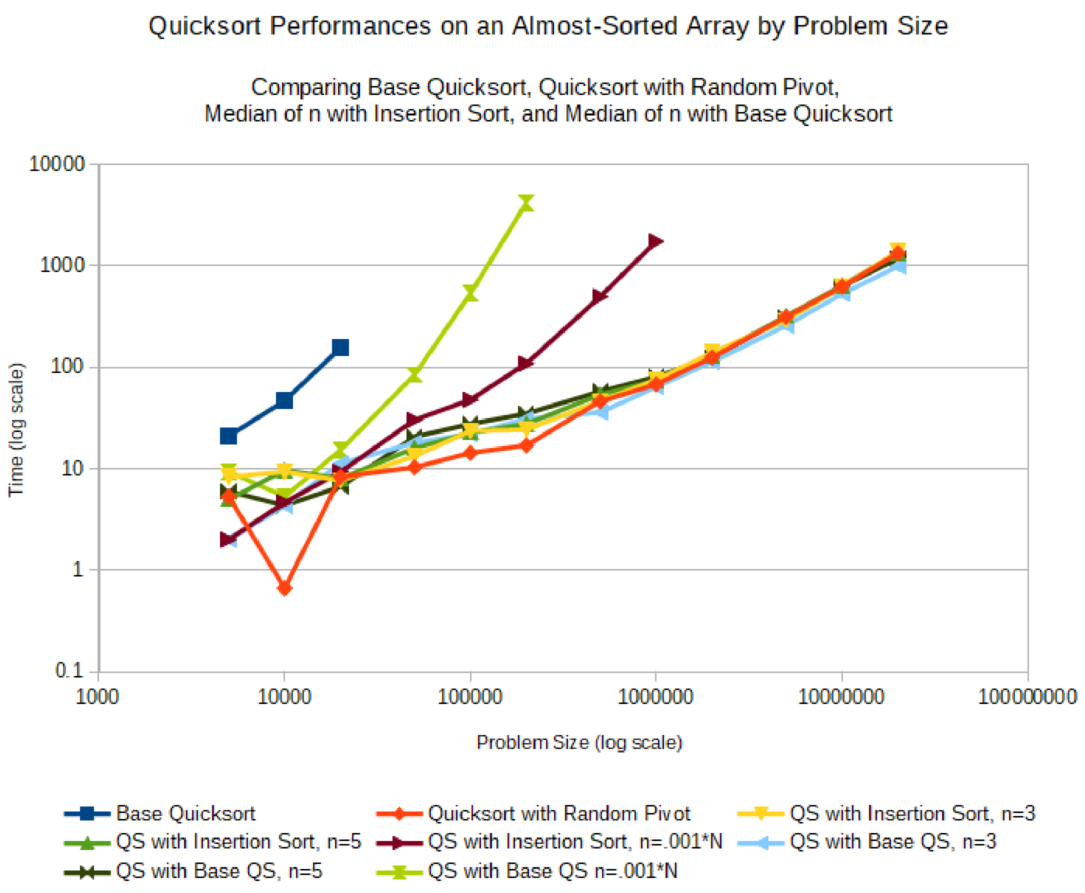
I found it interesting that quicksort with median of n using base quicksort performed about as well as quicksort with median of n using insertion sort for most problem sizes and outperformed it on large problems. Although quicksort with insertion sort uses an n^2 sort to get the medians as opposed to $\lg(n) * n$, I expected it to perform better than quicksort with quicksort because quicksort with quicksort has a deeper call stack (for $n = 5$, the call to getMedian will call 3 more levels of quicksort, whereas quicksort with insertion sort only has to call insertion sort once), and because n is small and fixed, the contribution to the overall runtime from the sort should not be very different for different sorting methods. Quicksort with insertion sort also had a built-in advantage over quicksort with quicksort:

Quicksort with insertion sort sorted the values in A as well as the values in medianIndices during the sort step within the partition call, whereas quicksort with quicksort only sorted the values in medianIndices.

I tried quicksort with median of n and quicksort using $n = 0.001N$, or one thousandth of the size of the *full* array because I had the idea that different n may be optimal for different problem sizes. (A bad partition in an array of size 1,000,000 is far worse than a bad partition in an array of size 5.) This didn't work well because quicksort calls partition on every problem size from N down to 2; a very large and a very small problem will *both* call quicksort on small problems, because it is recursive, and they will all use the same n on those small problems. So using $n = 0.001N$ set unreasonably high n for large problems and performed similarly to $n = 3$ on smaller problems. A better strategy may have been to set n depending on $r - p$, the width of the subarray to be partitioned, adapting to each subproblem. Another strategy could be to choose whether or not to use the median of n strategy at all based on the size of the subproblem and use the original partition algorithm (without median of n) unles the subproblem was very large, to balance the importance of good partitions on larger subarrays while still limiting the size of the call stack and avoiding excess overhead from calling getMedian on many very small subproblems.

Figure 3 compares runtimes for these algorithms on different problem sizes.

Figure 3: Comparison of different quicksort variants



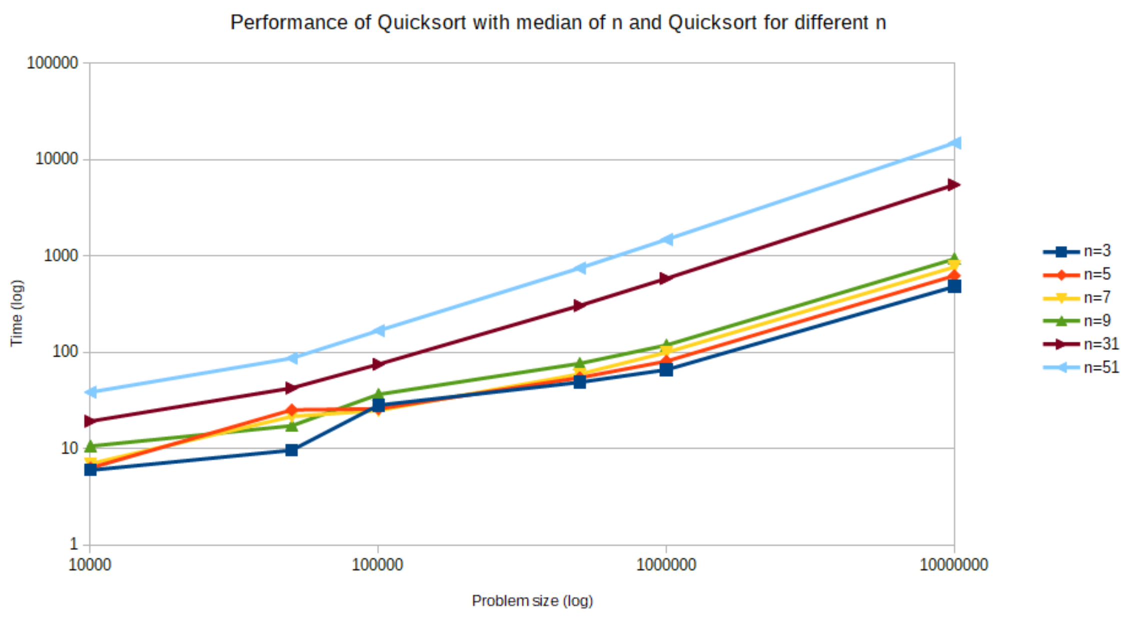
I also wanted to compare many different values for n directly. I took quicksort with median of n using quicksort and timed its performance on large problems with different values of n. The results are reported in Table 8 and shown in Figure 4.

N	$n = 3$	$n = 5$	$n = 7$	$n = 9$	$n = 31$	$n = 51$
10,000	6	6.3	7	10.7	19.3	38.7
50,000	9.7	25.3	21.7	17.3	42.7	87
100,000	28.3	26	25	36.7	75.3	168
500,000	49	54.7	59.7	76.7	306	750
1,000,000	66	81	100	119	585	1485
10,000,000	482	626	770	934	5495	14098

Table 8: Timing Comparison for Different n

Table 8 shows that $n = 3$ has the best performance for nearly every problem size. Performance was similar for $n = 3$, 5, 7, and 9, but clearly was worsening with larger n, as shown by $n = 31$ and $n = 51$. (Note that $n = 2k$ and $n = 2k + 1$ are almost computationally equivalent because of the floor operation used in returning the median; see Table 6.)

Figure 4: Comparison of quicksort with median of n for different n



Lastly, I investigated different levels of sorted-ness for the almost-sorted list by varying the variables random1 and random2 (see Table 4) to see if I could find any cases where $n = 5$ outperformed $n = 3$. $n = 3$ ran faster than $n = 5$ on a 20-million item array for all combinations of random1 and random2 I tried: 7 and 3, 9 and 4, 101 and 50, and 1001 and 500. I also tried creating autocorrelated lists where the correlation was not monotonic. They contained runs of 500 numbers where $A[i]$ was usually greater than $A[i+1]$ followed by runs of 500 numbers where $A[i]$ was usually less than $A[i-1]$; in other words, I populated A with sub-arrays of size 500 where the first array was almost-sorted in ascending order, the second in descending order, etcetera. Even these could not cause median of 5 to outperform median of 3. I did not test lists with many repeated elements; most of these lists likely had no duplicate elements because they were using randomly generated floating-point numbers. It is possible that there ways of generating arrays with repeated elements where $n = 5$ would perform better than $n = 3$. However, among almost-sorted lists, I found that $n = 3$ outperformed $n = 5$ in every situation I tested for large problem sizes.

5 Appendix: Code

Included in this Appendix is the Java code I used for quicksort with median of n and insertion sort, quicksort wih median of n and quicksort, and to generate almost-sorted lists.

5.1 Quicksort With Median of n and Insertion Sort

```
public class QuickSortI {

  static final int N=1000000;
  static double A[]=new double[N];
  static final int n0 = (int)(0.001*(double)N);
  static int[] medInds = new int[n0];
  static final boolean almostSorted = true;
  static boolean printArray = false;

  public static void Quicksort(int p, int r){
    if(p < r){
      int q = Partition(p, r);
      Quicksort(p, q-1);
      Quicksort(q+1, r);
    }
  }

  public static int getMedian(int p, int r, int n){
    if(p == r){
      return p;
    }
    //Populate indices
    for(int i = 0; i < n; i++) {
      medInds[i] = p + (i)*((r-p)/(n-1));
    }

    //Sort to find the index with the median value out of the three indices
    //medKeys passed to sort by A[medInds[i]], not medInds[i].
    medInds = insertionSort(medInds);
    return(medInds[(int)Math.floor(n/2.0)]);
  }



  public static int[] insertionSort(int[] C) {
    int j;
    int i = 0;
    double key = 0;
    int indKey = 0;
    //For each item in B...
    for(j = 1; j < C.length; j++) {
      //The value by which we sort is the item in that index of A
      key = A[C[j]]; //Value from actual array at index C[j]
      indKey = C[j];
      i = j-1;
      while(i > -1 && A[C[i]] > key) {
        A[C[i+1]]=A[C[i]]; //We will actually sort these values in A[] as well
        C[i+1]=C[i];       //Sort C alongside
        i--;
      }
      A[C[i+1]] = key;
      C[i+1] = indKey;
    }
    return C;
  }

  public static boolean checkSorted() {
    //Returns true if the list is sorted and false otherwise.
    for(int i = 0; i < N-1; i++) {
      if(A[i] > A[i+1]) {
        return false;
      }
    }
    return true;
  }



  public static int Partition(int p, int r) {
    //find median of n0 with getMedian
    int medInd = getMedian(p, r, n0);
    //set x to median
    double x=A[medInd];
    int i = p-1;
    int j = 0;
    double hold;
    //Swap median to last position
    hold = A[r];
    A[r] = A[medInd];
    A[medInd]=A[r];

    for(j = p; j <= r-1; j++){

      if(A[j] <= x) {
        i++;
```

```
        hold = A[i];
        A[i] = A[j];
        A[j] = hold;
      }
    }
    hold = A[i+1];
    A[i+1] = A[j];
    A[j] = hold;

    return i+1;
  }
}
```

5.2 Quicksort With Median of n and Quicksort

```
public class QuickSortQS {

  static final int N=10000000;
  static double A[]=new double[N];
  static boolean almostSorted = true;
  static boolean printArray = false;
  static int n=51;
  static int C[] = new int[n]; //Used to hold indices


  //Implicit parameter = A
  public static void Quicksort(int p, int r){
    if(p < r){
      int q = Partition(p, r);
      Quicksort(p, q-1);
      Quicksort(q+1, r);
    }
  }

  //Implicit parameter = C
  public static void QuicksortC(int p, int r){
    if(p < r){
      int q = nonrecurPartition(p, r);
      QuicksortC(p, q-1);
      QuicksortC(q+1, r);
    }
  }

  //Partition, but doesn't call getMedian.
  //Also, implicit parameter = C.
  public static int nonrecurPartition(int p, int r) {
```

```
  double x=C[r];
  int i = p-1;
  int j = 0;
  int hold;

  for(j = p; j <= r-1; j++){

    if(C[j] <= x) {
      i++;
      hold = C[i];
      C[i] = C[j];
      C[j] = hold;
    }
  }
  hold = C[i+1];
  C[i+1] = C[j];
  C[j] = hold;

  return i+1;
}

//Get median. Called ONLY by Quicksort, NOT by QuicksortC.
public static int getMedian(int p, int r){
  if(r - p < 2){
    //If r-p < 2,
    //we don't have 3 separate numbers of which to find the median,
    //so just
    return r;
  }
  //Populate indices

  //Uncomment for testing
  //System.out.println("Call size " + (r-p) + ", unsorted:");
  int ind = 0;
  for(int i = 0; i < n; i++) {
    ind = p + (i)*(int)Math.max((r-p)/(n-1), 1); //If the fraction is too small,
    //just go up by 1 each time until we hit the limit of this interval (r).
    if(ind <= r) {
      C[i] = ind;
    } else {
      C[i] = r;
    }
    //for testing
    //System.out.println(C[i] + ": " + A[C[i]]);
  }
  //Sort to find the index with the median value out of the n indices
  QuicksortC(0, n-1);
  /*
  //For testing
  System.out.println("Sorted:");
  for(int i = 0; i < n; i++) {
    System.out.println(C[i] + ": " + A[C[i]]);
  }
  */
  return(C[(int)Math.floor(n/2.0)]);
}

//Returns true if A is sorted and false otherwise
public static boolean checkSorted() {
  for(int i = 0; i < N-1; i++) {
    if(A[i] > A[i+1]) {
      return false;
    }
  }
  return true;
}


//This is the partition that getMedian
public static int Partition(int p, int r) {
  int medInd = getMedian(p, r);
  double x=A[medInd];
  int i = p-1;
  int j = 0;
  double hold;
  //Swap median to last position
  hold = A[r];
  A[r] = A[medInd];
  A[medInd]=A[r];

  for(j = p; j <= r-1; j++){

    if(A[j] <= x) {
      i++;
      hold = A[i];
      A[i] = A[j];
      A[j] = hold;
    }
  }
  hold = A[i+1];
  A[i+1] = A[j];
  A[j] = hold;
```

```
    return i+1;
  }
}
```

5.3 Generating an Almost-Sorted List

The following code snippet populates an array A with an almost-sorted list. Varying the constants could give different degrees of sorted-ness.

```
      A[0] = (Math.random()*10)+1;
      for(int i = 1; i < A.length; i++){
        A[i] = A[i-1] + (Math.random()*12) - 2;
      }
```