InSight: RIVIER ACADEMIC JOURNAL, VOLUME 7, NUMBER 1, SPRIING 2011

AN ANALYSIS OF THE WORST-CASE PERFORMANCE OF QUICKSORT

Robert R. Marceau*
M.S. Program in Computer Science, Rivier College

Abstract

C.A.R. Hoare's quicksort algorithm has become a very popular sorting algorithm due to the average performance of $\Theta(n \log n)$, limited use of extra storage (typically $\Theta(\log_2 n)$ recursive calls) and better performance on average compared to heapsort (another $\Theta(n \log n)$ sorting algorithm). It may be found in several standard libraries supporting C, C++, and Java. The major drawback in the quicksort algorithm is the $\Theta(n^2)$ worse case performance. Unfortunately, this performance is exhibited for some rather common initial permutations. The author intends to look into this performance of the quicksort algorithm, and in particular potential modifications to minimize the probability that the worst-case performance will be exhibited.

1 Historical Note

The original quicksort was described by C.A.R. Hoare in Algorithms 63 and 64 of the Collected Algorithms from the Association for Computing Machinery. (Presented in the original Algol). [1]

1.1 Algorithm 63 - partition

```
procedure partition (A,M,N,I,J); value M,N;
      array A; integer M,N,I,J;

comment: I and J are output variables, and A is the array (with subscript bounds
M:N) which is operated upon by this procedure.

Partition takes the value X of a random element of the array A, and rearranges the
values of the elements of the array in such a way that there exist integers I and J
with the following properties:

      M <= J < I <= N provided M < N
      A[R] <= X for M <= R <= J
      A[R] = X for J < R > I
      A[R] >= X for I <= R <= N

The procedure uses an integer procedure random (M,N) which chooses equiprobably a
random integer F between M and N, and also a procedure exchange, which exchanges
the values of its two parameters;

begin real X; integer F;
      F := random (M,N); X := A[F];
up: for I := 1 step 1 until N do
            if X < A[I] then go to down;
      I := N;
```


ISSN 1559-9388 (online version), ISSN 1559-9396 (CD-ROM version).

```
down: for J := J step -1 until M do
            if A[J] < X then go to change;
      J := M;
change: if I < J then begin exchange (A[I],A[J]);
                  I := I + 1; J := J - 1;
                  go to up;
            end
else if I < F then begin exchange (A[I],A[F]);
                  I := I + 1;
            end
else if F < J then begin exchange (A[F],A[J]);
                  J := J - 1;
            end;
end partition
```

1.2 Algorithm 64 - quicksort

```
procedure quicksort (A,M,N); value M,N;
      array A; integer M,N;

comment: Quicksort is a very fast and convenient method of sorting an array in the
random-access store of a computer. The entire contents of the store may be sorted,
since no extra space is required. The average number of comparisons made is 2(M-N)*
ln(N-M), and the average number of exchanges is one sixth this amount. Suitable
refinements of this method will be desirable for its implementation on any actual
computer;

begin integer I,J;
      if M < N then begin partition(A,M,N,I,J);
                  quicksort(A,M,J);
                  quicksort(A,I,N);
            end
end quicksort
```

2 Java Implementation

A Java implementation of Quicksort was created and instrumented to count the number of key comparisons and the number of key exchanges. The source code is included in the appendixes. The original Java implementation uses the first element of the subarray as the pivot value, as described in [2] pages 159-171.

3 Average Case Performance

In the average case, quicksort has a recurrence relation of $T(n) = 2T(n/2)$. That is, on average, the pivot procedure produces two subarrays of approximately n/2 elements. The depth of the recursion tree is $\log_2 n$. Summing over all the levels, we have $\Theta(n \log n)$. [2] pages 165-168, [3] pages 159-160, [4] pages 544-546, [5] pages 244-245

4 Worse Case Performance

Quicksort has a $\Theta(n^2)$ worse case performance. [2] pages 162-165, [3] page 156, [4] page 547 [5] page 243

4.1 Realized when array is already in ascending sequence

In the case where the array is in ascending sequence, the partition procedure will partition the array such that the left subarray has only one element. Here the recursion relationship degrades to $T(n) = T(n-1) + \Theta(1)$. In this case, $T(n) \in \Theta(n^2)$.

4.2 Realized when array is in descending sequence

In the case where the array is in descending sequence, the partition procedure will partition the array such that the right subarray has only one element. Performance is also $\Theta(n^2)$, as in the previous case.

5 Near Worse Case Performance

Near worse case performance is realized when array is already in nearly ascending or descending sequence. A typical example would be a small set of elements (all with keys greater than the existing array) appended to the previously sorted array. The various version of quicksort were run with an array that had the first 90% of the elements in either ascending or descending sequence, followed by a set of either ordered or unordered elements with larger keys.

An occurrence of this is not uncommon, when an implementation naïvely appends the new elements with assigned identification numbers to an already existing array. A much better approach would be to sort the new elements separately and then merge the results with the existing array. (Since the new elements would have assigned identification numbers, these may be in a near ascending sequence, so the use of the first element as a pivot for a quicksort of the new elements would exhibit $\Theta(n^2)$ behavior which should be avoided).

Another situation that may occur is when the array is already sorted by one key and then sorted by another key that is not independent. Consider the case where the array is initially sorted by zip code and then quicksort is used to sort it by state. Since zip codes are grouped by state, the array will contain several long runs of keys. There is a similar dependence between social security numbers and state of residence when the number is assigned.

6 Avoidance of Worse Case and Near Worse Case Performance

6.1 Random selection

A randomly selected element in the subarray is exchanged with the first element and becomes the pivot element. This method was used by C.A.R. Hoare in his original implementation. [1] Algorithm 63

6.2 Median

The median of a small number of elements chosen from the subarray is exchanged with the first element and becomes the pivot element. [6] page 123

7 Alternative Method for Small Subarrays

Another quicksort optimization involves the use of an alternative sorting algorithm when the subarray size is below a certain limit. Typically, this limit is chosen as 2 or 3, in which case the elements may be ordered using a decision tree. Although this will not affect the asymptotic behavior, it will eliminate a few levels from the recursion tree and reduce stack usage. The reduction will be Θ(n), reducing the number of recursive calls by n, 3n/2 , 7n/4 in the cases where one, two, or three levels are eliminated.

8 Implementations with Various Improvements

The following implementations of the quicksort algorithm where written in Java (included in the appendixes) and run to gather data.

q0	Original version using the first element as the pivot
q1	Decision tree for n < 3
q2	Decision tree for n < 4
q3	Decision tree for n < 4, median of left, middle and right as pivot
q4	Decision tree for n < 4, random pivot selection

9 Results

Each implementation was executed 100 times for permutations of size 10 to 100 in steps of 10. Nine different types of permutations were used (all values were unique):

aa	A strictly ascending permutation
ad	The first 90% were ascending, 10% descending
ar	The first 90% were ascending, 10% random
da	The first 90% were descending, 10% ascending
dd	A strictly descending permutation
dr	The first 90% were descending, 10% random
ra	The first 90% were random, 10% ascending
rd	The first 90% were random, 10% descending
rr	A random permutation

Number of comparisons for q0

Permutation	10	20	30	40	50	60	70	80	90	100
aa	63	228	493	858	1323	1888	2553	3318	4183	5148
ad	63	229	494	860	1325	1891	2556	3322	4187	5153
ar	63	228	494	859	1324	1889	2553	3316	4178	5140
da	60	205	431	740	1130	1603	2157	2794	3512	4313
dd	68	238	508	878	1348	1918	2588	3358	4228	5198
dr	60	205	432	741	1131	1604	2157	2792	3507	4305
ra	53	137	233	330	436	551	668	796	918	1042
rd	53	138	232	338	437	560	679	799	921	1053
rr	54	138	235	340	448	570	680	794	919	1052

Table 1: Original version using the first element as the pivot

Number of comparisons for q1

Permutation	10	20	30	40	50	60	70	80	90	100
aa	61	226	491	856	1321	1886	2551	3316	4181	5146
ad	61	226	492	857	1323	1888	2554	3319	4185	5150
ar	61	226	491	856	1320	1884	2547	3310	4171	5132
da	56	200	427	735	1126	1598	2153	2789	3508	4308
dd	65	235	505	875	1345	1915	2585	3355	4225	5195
dr	56	200	427	735	1125	1596	2150	2782	3498	4294
ra	45	121	213	301	401	508	618	741	856	972
rd	45	122	211	309	402	517	630	742	858	983
rr	46	122	212	309	410	523	626	732	849	976

Table 2: Decision tree for n < 3

Number of comparisons for q2

Permutation	10	20	30	40	50	60	70	80	90	100
aa	58	223	488	853	1318	1883	2548	3313	4178	5143
ad	58	224	489	855	1320	1886	2551	3317	4182	5148
ar	58	223	488	854	1318	1881	2544	3307	4166	5127
da	53	195	421	730	1120	1593	2147	2784	3502	4303
dd	63	233	503	873	1343	1913	2583	3353	4223	5193
dr	53	195	421	731	1120	1591	2143	2776	3490	4287
ra	40	111	197	282	377	479	586	703	815	927
rd	40	112	196	290	377	490	597	707	816	938
rr	41	112	197	289	385	494	591	693	807	927

Table 3: Decision tree for n < 4

Number of comparisons for q3

Permutation	10	20	30	40	50	60	70	80	90	100
aa	39	101	168	247	322	396	484	577	662	744
ad	39	102	170	250	325	396	484	578	667	749
ar	39	101	169	247	325	399	487	581	669	752
da	50	124	208	297	395	484	578	682	792	895
dd	41	106	169	251	323	406	490	584	664	755
dr	50	124	208	300	397	491	585	690	800	906
ra	44	115	201	290	381	480	581	690	789	893
rd	42	117	203	294	387	481	582	684	802	899
rr	46	119	205	299	393	495	602	700	811	933

Table 4: Decision tree for $n < 4$, median of left, middle and right as pivot

Number of comparisons for q4

Permutation	10	20	30	40	50	60	70	80	90	100
aa	40	112	195	277	369	471	569	673	782	883
ad	40	113	192	276	375	476	567	680	778	892
ar	41	112	189	279	370	470	568	675	775	895
da	41	113	192	283	385	471	571	684	786	887
dd	42	111	195	280	368	476	588	676	781	888
dr	40	111	197	282	375	473	571	678	789	886
ra	42	115	192	289	378	476	578	672	778	888
rd	41	113	194	288	383	479	578	685	791	890
rr	43	113	195	286	378	470	579	681	801	902

Table 5: Decision tree for $n < 4$, random pivot selection

As can be seen in the tables above, q0, q1, and q2 exhibit a worst case performance of $\Theta(n2)$ for permutations that are ordered or nearly ordered. The differences between q0, q1, and q2 for the ordered permutations (aa, dd) is small. This is consistent with decision tree only being used once for the final partitioning of the left or right subarray. For the nearly ordered permutations (ad, ar, da, and dr) the difference is also small, since the decision tree is only used for a small number of leaf nodes. In these cases, the recurrence relation degrades to $T(n) = T(n-1) + \Theta(1)$, with each partitioning producing subarrays lengths 1 and $n-1$. This will continue until $n-1 = 3$ for q1 or $n-1 = 4$ in the case of q2, at which point the decision tree will be utilized in place of further partitioning.

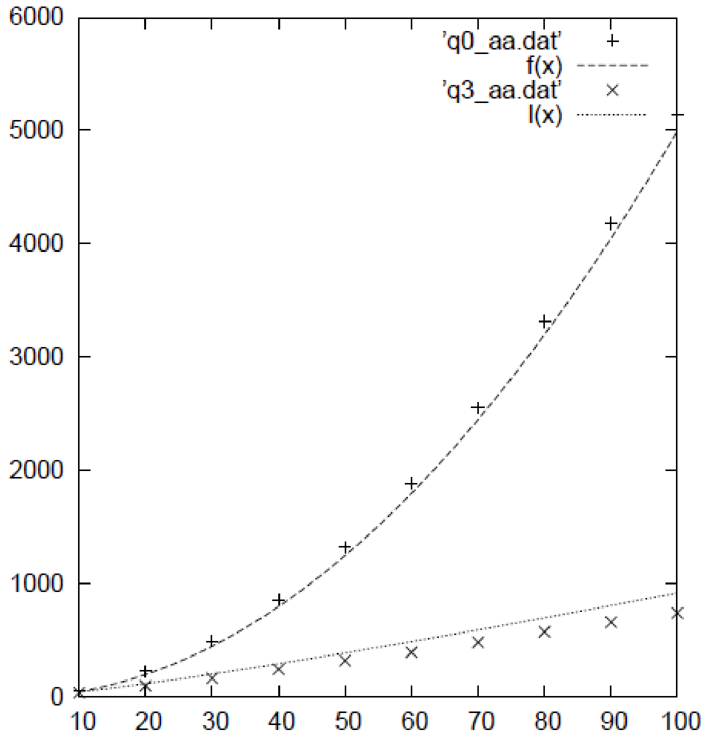

Figure 1: Plot of number of comparisons vs. data set size, strictly ascending Original (q0) and Median pivot (q3); $f(x) = ½x^2$ and $l(x) = 2x\log_2 x$

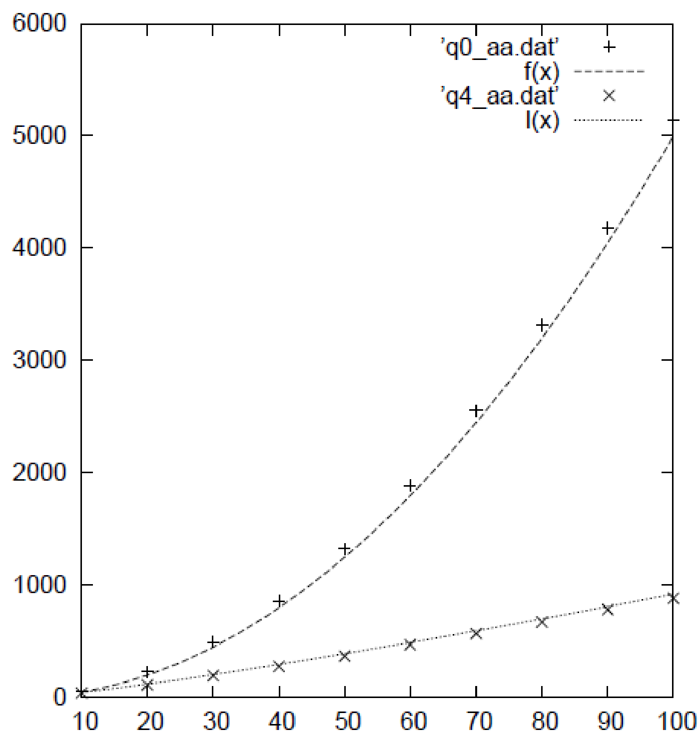

Figure 2: Plot of number of comparisons vs. data set size, strictly ascending Original (q0) and Random pivot (q4); $f(x) = ½x^2$ and $l(x) = 2x\log_2 x$

For random permutations, the use of a decision tree in q1 and q2 makes a slightly larger improvement over the performance of q0. The performance of all implementations for random permutations is $\Theta(n \log n)$, as expected.

The use of either the median (in q3) or a randomly selected element (in q4) as the pivot value reduces the worst case performance from $\Theta(n^2)$ to $\Theta(n \log n)$ as can be seen in Figures 1 and 2. In each graph, two reference function, $f(x) = ½x^2$ and $l(x) = 2x\log_2 x$ are also plotted. The constants were chosen to fit the actual data. It is also noteworthy to observe that q3 and q4 performed as well or a little better than q2 (which uses the same size decision tree as q3 and q4) for the random permutations.

For cases where the permutation is mostly ascending (aa, ad, ar) q3 (using the median as pivot) showed about a 15% improvement over q4 (where a randomly selected element of the subarray is used as the pivot value).

In conclusion, it is apparent that using the median value for the pivot produces the best performance. For the random permutations, the median pivot performs within a few percent of the random pivot implementation. For ordered permutations, the median value pivot implementation performs best. Both q3 and q4 are $\Theta(n \log n)$ for all of the permutations examined, while q0, q1, and q2 were $\Theta(n \log n)$ only for the random permutations and had $\Theta(n^2)$ behavior for the permutations that were either ordered or nearly ordered. ■

References

[1] ACM. Conference Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing: Papers Presented at the Symposium, Los Angeles, California, April 28-30, 1980. Association for Computing Machinery, 1998.

[2] Sara Baase and Allen Van Gelder. Computer Algorithms: Introduction to Design and Analysis (3rd Edition). Addison Wesley, 1999.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms (MIT Electrical Engineering and Computer Science). The MIT Press, 1990.

[4] Thomas A. Standish. Data Structures, Algorithms, and Software Principles in C. Addison Wesley, 1994.

[5] Mark A. Weiss. Data Structures and Algorithm Analysis in C (2nd Edition). Addison Wesley, 1996.

[6] Donald E. Knuth. The Art of Computer Programming: Sorting and Searching. Volume 3 (Addison Wesley Series in Computer Science and Information Processing). Addison-Wesley, 1973.

Appendix A: Original version using first element as pivot

```
//
// Plain version
//
import java.util.Scanner;
public class q0
{
	static int n_cmp = 0;
	static int n_xch = 0;

	static boolean isGreater (int x, int y)
	{
		n_cmp++;

		return x > y;
```

```
}

static boolean isLess (int x, int y)
{
        n_cmp++;

        return x < y;
}

static void xchg (int [] x, int i, int j)
{
        int t = x[i];
        n_xch++;

        x[i] = x[j];
        x[j] = t;
}

public static void quicksort (int[] x)
{
        quicksort(x, 0, x.length - 1);
}

public static void quicksort (int[] x, int p, int r)
{
        if (r <= p)
                return;

        int q = partition (x, p, r);
        quicksort (x, p, q);
        quicksort (x, q+1, r);

        return;
}

public static int partition (int[] x, int p, int r)
{
        int xx = x[p];
        int i = p - 1;
        int j = r + 1;

        for (;;) {
                do {
                        j--;
                } while (isGreater(x[j], xx));
                do {
                        i++;
                } while (isGreater(xx, x[i]));
                if (i < j) {
                        xchg (x, i, j);
                } else
                        return j;
        }
}
```

```
	public static void print_array (int[] x)
	{
		final int n = x.length;

		for (int i = 0; i < n; i++)
			System.out.printf ("%5d%c",
				x[i], ((i+1)%10) == 0 ? '\n': ' ');
		System.out.printf ("\n");
	}

	public static void main (String[] args)
	{
		Scanner input = new Scanner(System.in);

		int n = Integer.parseInt(args[0]);
		int [] x = new int [n];
		int t = 0;

		while (input.hasNext()) {

			for (int i = 0; i < n; i++)
				x[i] = input.nextInt();

			quicksort (x);
			t++;
		}
		if (t != 0)
			System.out.printf ("%d", n_cmp/t);
	}
}
```

Appendix B: Decision tree for less than three elements

```
//
// Less than 3 by decision tree
//
import java.util.Scanner;
public class q1
{
	//
	// Code identical to q0 omitted
	//
	public static void quicksort (int[] x, int p, int r)
	{
		if (r <= p)
			return;

		if (r - p < 2) {
			if (isGreater(x[p], x[r]))
				xchg (x, p, r);
			return;
		}
```

```
		int q = partition (x, p, r);
		quicksort (x, p, q);
		quicksort (x, q+1, r);

		return;
	}

	public static int partition (int[] x, int p, int r)
	{
		int xx = x[p];
		int i = p - 1;
		int j = r + 1;

		for (;;) {
			do {
				j--;

			} while (isGreater(x[j], xx));
			do {
				i++;
			} while (isGreater(xx, x[i]));
			if (i < j) {
				xchg (x, i, j);
			} else
				return j;
		}
	}
	//
	// Code identical to q0 omitted
	//
}
```

Appendix C: Decision tree for less than four elements

```
//
// Less than 4 by decision tree
//
import java.util.Scanner;
public class q2
{
	//
	// Code identical to q0 omitted
	//
	public static void quicksort (int[] x, int p, int r)
	{
		if (r <= p)
			return;

		if (r - p == 2) {
			if (isLess(x[p], x[p+1])) {
				if (isLess(x[p+1], x[r]))
					return;
				if (isLess(x[p], x[r])) {
					xchg (x, p+1, r);
```

```
                        } else {
                                xchg (x, p+1, r);
                                xchg (x, p, p+1);
                        }
                } else {
                        if (isLess(x[p], x[r])) {
                                xchg (x, p, p+1);
                        } else {
                                if (isLess(x[p+1], x[r])) {
                                        xchg (x, p, r);
                                        xchg (x, p, p+1);
                                } else {
                                        xchg (x, p, r);
                                }
                        }
                }
                return;
        }

        if (r - p < 2) {
                if (isGreater(x[p], x[r]))
                        xchg (x, p, r);
                return;
        }

        int q = partition (x, p, r);
        quicksort (x, p, q);
        quicksort (x, q+1, r);

        return;
}

public static int partition (int[] x, int p, int r)
{
        int xx = x[p];
        int i = p - 1;
        int j = r + 1;

        for (;;) {
                do {
                        j--;

                } while (isGreater (x[j], xx));
                do {
                        i++;
                } while (isGreater(xx, x[i]));
                if (i < j) {
                        xchg (x, i, j);
                } else
                        return j;
        }
}
//
// Code identical to q0 omitted
```

```
	//
}
```

Appendix D: Less than 4 by decision tree, median pivot

```
//
// Less than 4 by decision tree, median(1, p/2, p) pivot
//
import java.util.Scanner;
public class q3
{
	//
	// Code identical to q0 omitted
	//
	static int median (int [] x, int p, int q, int r)
	{
		if (isLess(x[p], x[q])) {
			if (isLess(x[q], x[r]))
				return q;
			if (isLess(x[p], x[r])) {
				return r;
			} else {
				return p;
			}
		} else {
			if (isLess(x[p], x[r])) {
				return p;
			} else {
				if (isLess(x[q], x[r])) {
					return r;
				} else {
					return q;
				}
			}
		}
	}
	public static void quicksort (int[] x)
	{
		quicksort(x, 0, x.length - 1);
	}

	public static void quicksort (int[] x, int p, int r)
	{
		if (r <= p)
			return;

		if (r - p == 2) {
			if (isLess(x[p], x[p+1])) {
				if (isLess(x[p+1], x[r]))
					return;
				if (isLess(x[p], x[r])) {
					xchg (x, p+1, r);
				} else {
					xchg (x, p+1, r);
```

```
                        xchg (x, p, p+1);
                    }
                } else {
                    if (isLess(x[p], x[r])) {
                        xchg (x, p, p+1);
                    } else {
                        if (isLess(x[p+1], x[r])) {
                            xchg (x, p, r);
                            xchg (x, p, p+1);
                        } else {
                            xchg (x, p, r);
                        }
                    }
                }
                return;
            }

            if (r - p < 2) {

                if (isGreater(x[p], x[r]))
                    xchg (x, p, r);
                return;
            }

            int q = partition (x, p, r);
            quicksort (x, p, q);
            quicksort (x, q+1, r);

            return;
        }

        public static int partition (int[] x, int p, int r)
        {
            int xx;
            int i = p - 1;
            int j = r + 1;

            if (r - p >= 4) {
                int m = median (x, p, p+(r-p)/2, r);

                if (m != p)
                    xchg (x, p, m);
            }

            xx = x[p];

            for (;;) {
                do {
                    j--;

            } while (isGreater(x[j], xx));
            do {
                i++;
            } while (isGreater(xx, x[i]));
```

```
		if (i < j) {
			xchg (x, i, j);
		} else
			return j;
		}
	}
	//
	// Code identical to q0 omitted
	//
}
```

Appendix E: Less than 4 by decision tree, random pivot

```
//
// Less than 4 by decision tree, random pivot
//
import java.util.Scanner;
public class q4
{
	//
	// Code identical to q0 omitted
	//

	public static void quicksort (int[] x, int p, int r)
	{
		if (r <= p)
			return;

		if (r - p == 2) {
			if (isLess(x[p], x[p+1])) {
				if (isLess(x[p+1], x[r]))
					return;
				if (isLess(x[p], x[r])) {
					xchg (x, p+1, r);
				} else {
					xchg (x, p+1, r);
					xchg (x, p, p+1);
				}
			} else {
				if (isLess(x[p], x[r])) {
					xchg (x, p, p+1);
				} else {
					if (isLess(x[p+1], x[r])) {
						xchg (x, p, r);
						xchg (x, p, p+1);
					} else {
						xchg (x, p, r);
					}
				}
			}
			return;
		}

		if (r - p < 2) {
```

```
                if (isGreater(x[p], x[r]))
                    xchg (x, p, r);
                return;
            }

            int q = partition (x, p, r);
            quicksort (x, p, q);
            quicksort (x, q+1, r);

            return;
        }

        public static int partition (int[] x, int p, int r)
        {
            int xx;
            int i = p - 1;
            int j = r + 1;

            int m = (int) ((r-p+1)* Math.random()) + p;

            if (m != p)
                xchg (x, p, m);

            xx = x[p];

            for (;;) {
                do {
                    j--;

                } while (isGreater(x[j], xx));
                do {
                    i++;
                } while (isGreater(xx, x[i]));
                if (i < j) {
                    xchg (x, i, j);
                } else
                    return j;
            }
        }
        //
        // Code identical to q0 omitted
        //
}
```

---

* **ROBERT MARCEAU** wrote his first computer program in October, 1969 on a DEC PDP-8/I running TSS/8. Since that time, he has earned a BS in Mathematics from the University of Massachusetts-Lowell in 1977. After spending the next thirty years in the software industry, he has returned to the University of Massachusetts-Lowell to complete his MS in Mathematics (expected May, 2011) and has started the MS in Computer Science program at Rivier College in Nashua, NH. He is currently an Adjunct Faculty member at Nashua Community College and teaching Fundamentals of Operating Systems and Object Oriented Programming with C++.