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:

$$M \leq J < I \leq N \text{ provided } M < N$$
$$A[R] \leq X \text{ for } M \leq R \leq J$$
$$A[R] = X \text{ for } J < R > I$$
$$A[R] \geq X \text{ for } I \leq R \leq N$$

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;

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
544-546, [5] pages 244-245

4 Worse Case Performance

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 naively 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 $\Theta(n)$, 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

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

Table 1: Original version using the first element as the pivot
AN ANALYSIS OF THE WORST-CASE PERFORMANCE OF QUICKSORT

Number of comparisons for q1

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

Table 2: Decision tree for n < 3

Number of comparisons for q2

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

Table 3: Decision tree for n < 4
As can be seen in the tables above, q0, q1, and q2 exhibit a worst case performance of Θ(n^2) 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 \( T(n) = n \Theta(n) + \Theta(1) \), 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.
AN ANALYSIS OF THE WORST-CASE PERFORMANCE OF QUICKSORT

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


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;
    }
AN ANALYSIS OF THE WORST-CASE PERFORMANCE OF QUICKSORT

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;
        }
AN ANALYSIS OF THE WORST-CASE PERFORMANCE OF QUICKSORT

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);
                }
            }
        }
    }
}
```java
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
```
AN ANALYSIS OF THE WORST-CASE PERFORMANCE OF QUICKSORT

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

import java.util.Scanner;

public class q3 {
    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]));
    }
}
AN ANALYSIS OF THE WORST-CASE PERFORMANCE OF QUICKSORT

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) {
            // Code identical to q0 omitted
        }
    }
}
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
//