Quicksort

Def^n Quicksort (a sequence) -- Recursive def^n

```
Qsort [ ] = [ ]
Qsort a : X  =   Qsort [ b| b ← X ; b ≤a]
                     ++ [a] ++
                 Qsort [ b| b ← X ; b > a]
```

The list a : X (“a prepended the X”) is partitioned about a and each partition is sorted.

e.g. Qsort [3,1,4,1,5,9,2] = Qsort [1,1,2] ++ [3] ++ Qsort [4,5,9]
= [1,1,2] ++ [3] ++ [4,5,9]
= [1,1,2,3,4,5,9]

Not^n:

[y | y ← T ; P(y)] is the sequence of all y drawn from T, satisfying P.
e.g. L = [y | y ← T ; y ≤ x] is the sequence of elements in T ≤ x

“joining” two sequences, L and M.
We use the notation L ++ M for joining the sequence M onto the end of L.

Quicksort an Array

Based on the above def^n of Quicksort we want a algorithm for sorting arrays in-place.
A simple example of a Root/Test class for Quicksort is:

```
class
      SORTROOT
creation
      make
feature
      make is
            local
                  s : QUICKSORT[INTEGER]
                  a : ARRAY[INTEGER]
                  i : INTEGER
            do
                  !!a.make(1,7)
                  a := <<3,1,4,1,5,9,2>>
                  !!s
                  s.sort(a, a.lower, a.upper)
                  -- Print out sorted array
                  from
                        i := a.lower
                  until
                        i > a.upper
                  loop
                        io.put_integer(a.item(i))
                        io.put_character(‘ ‘)
                        i := i+1
                  end
                  io.new_line
            end—make
end—SORTROOT
```

Top Level view of Quicksort Algorithm on Arrays:

Given an array A, partition A into 2 sections such that all the items in the left section are less than or equal to (≤ "at most") the items in the right section. The array A is partitioned about an item, P—the pivot. Having partitioned the array, each section is recursively (quick)sorted.

In more detail,

Step 1. Partition:

- Select an item in A for the pivot p,
- Scan from the left until A@i ≥ p
- Scan from the right until A@j ≤ p
- Exchange A@i and A@j
- Continue until scans meet / crossover

Step 2. Sort each section:

Having partitioned the array, we then recursively sort each section.

Eiffel Procedure for Partitioning

Given an array A and two 'markers' L and R, for determining the position of the split into sections, the procedure is as follows:

```
Partition (L0,R0 : INTEGER; p : G) is
    do
        from
            L := L0
            R := R0
        until
            L > R
        loop
            Left_Scan (p) -- Scan from the Left
            Right_Scan(p) -- Scan from the Right
            if L <= R then
                exchange(L,R) -- exchange A@L and A@R
                L := L+1
                R := R-1
            end
        end
    end—Partition
```

Note:

Even though A@L and A@R may be equal, they are still exchanged. A check for equality could be made but overall this would be more costly.

The procedures for Left_Scan and Right_Scan:

```
Left_Scan (p : G) is
        do
                from
                until
                        A.item(L) >=p
                loop
                        L := L+1
                end
        end—Left_Scan

Right_Scan (p : G) is
        do
                from
                until
                        A.item(R) <= p
                loop
                        R := R-1
                end
        end—Right_Scan
```

Note:

Scanning will halt if an item equal to p is found. The item P may appear more than once. If the array contained all equal items (and so all equal to the pivot) then in the above algorithm, the items in the array would be continually exchanged. This version of quicksort is not stable.

QuickSort

```
Qsort (Left, Right : INTEGER) is
        local
                i,j : INTEGER
                Pivot : G
        do
                if Left < Right then
                        Pivot := A.item((Left + Right)//2)
                        Partition(Left,Right, Pivot)
                        i := L
                        j := R
                        Qsort(Left,j)
                        Qsort(i,Right)
                end
        end—Qsort
```

The Sort procedure to be exported is:

```
sort(A0 : ARRAY [G]; lower, upper : INTEGER) is
        do
                A := A0
                Qsort(lower,upper)
        end—sort
```

The Class Quicksort

```
class QUICKSORT [G -> COMPARABLE]
feature     --Only ‘sort’ is exported
        sort(A0 : ARRAY [G]; low, high : INTEGER) is
                do
                        A := A0
                        Qsort(low, high)
                end—sort

feature {NONE}
        A : ARRAY [G]
        L, R : INTEGER

        exchange(i,j:INTEGER) is
                local
                        it : G
                do
                        it := A.item(i)
                        A.put(A.item(j), i)
                        A.put(it,j)
                end—exchange

Qsort (Left, Right : INTEGER) is
        local
                i,j : INTEGER
                Pivot : G
        do
                if Left < Right then
                        Pivot := A.item((Left + Right)//2)
                        Partition(Left,Right, Pivot)
                        i := L
                        j := R
                        Qsort(Left,j)
                        Qsort(i,Right)
                end
        end—Qsort
```

```
Partition (L0,R0 : INTEGER;  p : G) is
        do
                from
                        L := L0
                        R := R0
                until
                        L > R
                loop
                        Left_Scan (p)
                        Right_Scan(p)
                        if L <= R then
                                exchange(L,R) -- exchange items A @ L and A @ R
                                L := L+1
                                R := R-1
                        end
                end
        end—Partition

Left_Scan (p : G) is
        do
                from
                until
                        A.item(L) >= p
                loop
                        L := L+1
                end
        end—Left_Scan

Right_Scan (p : G) is
        do
                from
                until
                        A.item(R) <= p
                loop
                        R := R-1
                end
        end—Right_Scan

end—QUICKSORT
```

Quicksort Discussion

General Comments about sorting

In practice, sorting deals with records with keys and associated data. Large records may be costly to move or swap. This moving may be alleviated by having “pointers” from the keys to the the data. For clarity of exposition we will assume our arrays have just “keys” or items that can be compared.

Quicksort

Quicksort is regarded as the fastest method in general for sorting large arrays. Strictly speaking, Quicksort is not an “in-place” sort as the recursive calls require “stack-space” related to the array size. But according to Knuth, the “stack” will generally not be more than O(log n).

In very rare situations, Quicksort may not be O(n*(log n)) as the worst case running time is O(n^2).

Worst Case for Quicksort

The performance of Quicksort depends on how balanced the partitioning is.

The worst case is where on each partition the array is split into n-1 and 1 element regions. In the worst case, this extreme unbalanced partition happens every time. In a sense Quicksort then becomes “Select-Sort”.

Also, in the Quicksort algorithm, the left split is sorted first, due the order of the recursive calls. If the split is such that one item is always in the right split then we will need n recursive calls which will cause a the recursion stack to be size O(n). This defeats our assumption of Quicksort as an in-place sort. To overcome this, one could choose the smallest split to be recursively quicksorted.

The worst case scenario is extremely rare, since the pivot for partition is, in effect, a random element. Even in the case where the split is of constant proportions
e.g. the array is always split in the proportion of, say 1/100 and 99/100, the performance of Quicksort will still be O(n*log n).

Quicksort performs well even when Partition produces a mix of “good” and “bad” splits. If every second split is a worse case split the performance is still O(n*log n).

Item Moves/Exchanges by Partition

From the Partition algorithm, we see that having selected the pivot we sweep the entire array comparing items. So n comparisons are performed by each partition. On average, log n calls to Quicksort are performed and so the total number of comparisons is about n*log n.

We now consider the number of exchanges (or swaps) by partition. Without loss of generality we can assume that we have no repeated items and the items are the integers

1..n. (See Wirth “Algorithms and Data Structures” Ch 2.3.3). After partitioning about pivot k, the “expected” or mean number of exchanges is the number of items in the left split, k-1, multiplied by the probability of the item being exchanged. An item is exchanged if it bigger than the pivot and this exchange involves swapping with a smaller than pivot item.

During the partition an item is swapped if it is $\geq k$
tf. Prob of exchange = (n - (k-1)) / n
tf. #exchanges in partition = (k-1) * (n- (k-1)) / n

Expected number of exchanges

= (Summation over all n choices of pivot) divided by n

= 1 / n * (Sum k | k in 1..n : (k-1)((n-k+1) / n))

= n / 6 - 1 / 6n -- not an easy calculation

tf. # Exchanges ≈ n / 6

With optimal split, where the median (the middle item in size) is chosen as pivot we get
#Exchanges in Quicksort is (n / 6) * log n
From practise we get for Quicksort

#Comparisons = 1.4 n*log n (1.4 = 2*log 2)

#Exchanges = 1.4*(n / 6) * log n

Time complexity of Quicksort.

When a list on items is partitioned, the whole array is scanned just once. With exchanges and comparisons the time complexity is O(n). Let us approximate this to just n for simplicity.

The time complexity for Quicksort is

T(n) = n (for partition) + average time for quicksorting each split of size s and n-s

Assume no repeated items and that choice of pivot is random i.e. each item has chance of being the pivot. The size s depends on choice of pivot and so s can vary from 1 to n-1. Let us average over the choices of pivot.

T(n) = n + 1 / n * (Sum s | s in 1..n-1 : T(s) + T(n-s)) but
(Sum s | s in 1..n-1 : T(s)) = (Sum s | s in 1..n-1 : T(n-s))

tf. T(n) = n + 2 / n * (Sum s | s in 1..n-1 : T(s))

We can show by induction that

T(n) is O(n*log n) i.e. T(n) $\leq$ k(n*log n)

n = 2 -- ok as T(2) = 2 + T(1)

= 2

and k(n*log n) = 2k -- log is to the base 2

i.e. for k $\geq$ 1 T(2) $\leq$ k(2*log 2)

Sorting of 2 items involves at most 2 comparisons and at most one exchange.

n>2: Assume true for $s < n$, $T(s) \le k(s*\log s)$

show $T(n) \le k(n*\log n)$

$T(n) = n + 2/n * (\text{Sum } s \mid s \text{ in } 1..n-1 : T(s))$

by induction hypothesis,

$T(n) \le n + 2/n * k*(\text{Sum } s \mid s \text{ in } 1..n-1 : s*\log s)$

but $\log s \le \log n$, $1 \le s \le n$
(log is a monotonic increasing function)

tf. $T(n) \le n + 2k/n * (\log n)(\text{Sum } s \mid s \text{ in } 1..n-1 : s)$

tf. $T(n) \le n + 2k/n * (\log n)*(n(n-1)/2)$

tf. $T(n) \le n + k(n-1) \log n$

i.e. T(n) is O(n*log n)

Selecting the Pivot

In our algorithm for partition we chose the middle item as the pivot. If we chose the first or last item, the worse case for Quicksort would be an initially sorted array. In choosing the middle item as pivot then an initially sorted array would be optimal i.e. n*log n.

Hoare suggests picking an item at random for the pivot; this would have the overhead of using a random number generator for each split.

The version presented above (from Dijkstra and Wirth) of quicksort is very efficient as the inner loops in Partition are minimal. The inner loops don't check for out of bounds in the array. This check is not needed as the pivot item acts as a 'sentinel' or boundry marker for the loop iteration. In this version of quicksort it is necessary to choose for the pivot an item in the array itself.

Improvements to Quicksort

Recursion removal

It is debatable whether recursion removal will make quicksort better. It may depend on how the language compiler handles recursion. Wirth indicates that it makes no difference while Sedgewick indicates it does.

Sort small arrays using “simpler” sorts

Quicksort is appropriate for large arrays but it performs well even for small arrays, with size > 20. Quicksort can be adapted to call simpler sorts
e.g. select sort, when the array size gets below, say, 20.