# 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]
```

**NWt:**

[ y | y ← T ; P(y) ] is tPe sequence Wf all  y drawn from T, satQsfyQng

e.g.        L = [y | y ≤  x ] is tPe sequence Wf elements Qn T

TPe list a : X (“a prepended tPe X”) is partitQoned about a and each partitQon is sorted.

“joQnQng” two sequences, L and M.

We use tPe nWtatQon L ++ M for joQnQng tPe sequence M onto tPe end Wf L.

e.g.   Qsor     =    Qsort [1

=    [1,1

=    [1,1

## *uicksort an Array*

ased on tPe above def$^n$ Wf Quicksort we want a algorithm for sortQng arrays QV-place.

simple example Wf a RoWt / Test class for Quicksort is:

**ass**

```
        SORTROOT
creatQon
        make
feature
        make is
                local
                        s : QUICKSORT[INTEGER]
                        a : ARRAY[INTEGER]
                        Q : INTEGER
                dW
                        !!a.make(1,7)
                                             ,2>>
                        !    !    s
                        s.sort(a, a.lower, a.upper)
                        -- PrQnt out sorted array
                        from
Q := a.lWwer
                        uVtQT
Q > a.upper
                        loWp
Qo.put_Qnteger(a.item(Q))
io.put_character(‘ ‘)
                                Q := i+1
                        end
                        Qo.new_line
                end
end—SORTROOT
```

**Top Level view of Quicksort Algorithm Wn Arrays:**

Given an a Arˆ A, partitiWn

out an item, P—the pivot. Having partitiWned the aTrayj -each sectiWn is recursively

(quick)sorted.

In more detail,

***Step 1. PartitiWn:***

- Select an item in A for the pivot p,
- Scan from the left until A@i ≥ p
- Scan from the right until A@j

**T h e**

```
Lsi e f t
    d
       mrof
         u          n          t
         l  A   p  .  oi   t   eo
            ILE1
         e              n
    e —Ien            n
R        si  i        g       h        t
    d
         f             r           o
         u n t i l
          ≺ )(Rtt i A  p
            RR1
         e              n
                       n

Qsort(Left, Right : INTEGER)

P i v o t : G
    d W
            P ot := Aiitem((Left + Right)//2)
            i := L
           Rj
            Qsort(Left, j)

    e n-dQ

sort(A 0            :  i  s   ARRAY
         A
```

## *The Class Quicksort*

```
class QUICKSORT [G -> COMPARABLE]
```

```
            from

                R := R0
            untiT
                L > R
            loop
                Left_Scan (p)
                RQght_Scan (p)
                i fL <= R then

                    L := L+1
                    R := R-1
                end

        end +Partition

    Left_Scan       (p is   :       G)
        do
            from

                A.iteU(L) >= p
            loop
                L := L+1
            end
        end—Left_Scan

    RQght_Scan (p : G) is
        do

            untiT
                A.iteU(R) <= p
            loop
                R := R-1

        end—RQght_Scan

    uend—QUICKSORT
```

untiT

```
        do

                L := L0
```

Partition (L0,R0 : INTEGER) is -- exchange items A @ L and A @ R

# Quicksort Discussion

## General

[illegible]

[illegible]

the keys to the the data. For clarity of expWsition we will assume our arrays Pave just
“keys” or items that can be compared.

## Quick1ort

Quick1ort

Strictly speaking, Quicksort is not an “in-place” 1 t as the recursive calls require “stack-

[illegible]

[illegible]

[illegible]

[illegible] $^2$).

## Wor1t Case for Quick1ort

The performance of Quick1ort depends on hWw balanced the partitioning is.

The case is where on each partition the array is split into n-1 and 1 element
regioVs. In the wor1t case, this extreme unbalanced partitioV Pappens every time. In a

[illegible]

Also, in the Quick1ort 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
need n recursive calls which will cause a the recursion stack to be size O(n). This defe
our assumption of Quick1ort as an in-place 1ort. To overcome this, one could chWose the
smalle1 split to be recursively quick1 ted.

The worst case scenario is extremely rare, since the pivot

random element. Even in the case w

e.g. the array is always split in the propW

entire array co

1 .

[illegible] i g g

[illegible] kapped if it is ≥

[illegible] = [illegible]

( k -

S u m m

* ( S u m k | k i n

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

shWw T(n)≤ k(n*log n)

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

by inductQon hypotPesis,

T(n) ≤ n + 2/n * k*(Sum s | s in 1..n-1 : s*log s)

but log s ≤ log n, 1 ≤ s ≤ n
(log is a monotonic increasing functQon)

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

tf. T(n) ≤ n + 2k/n * (log n)*(n(n-1)/2)

tf. T(n) ≤ n + k(n-1) log n i.e. T(n) is O(n*log n)

## SelectQng tPe Pivot

In our algorQtPm for partQtQon we chWse tPe middle item as tPe pivot. If we chW
first or last Qtem, tPe worse case for QuicSsort would be an inQtially sorted array. In
choosing tPe middle Qtem as pivot tPen an inQtially sorted array would be optQmal i.e.

Hoare suggests picSing an Qtem at random for tPe pivot, tPis would Pave tPe overPead o
TPe version presented above (from DijSstra and Feijen) of quicSsort is very efficient
as tPe inner loops in PartQtQon are minimal. TPe inner loops don't check for out of bWunds
in tPe array. TPis checS is Vot needed as tPe pivot Qtem acts as a 'sentQnel' or bWundry
marker for tPe loop iteratQon. In tPis versQon of quicSsort Qt is Vecessary to chWose for tPe
pivot Qtem in tPe array itself.

## ImprWvements to QuicSsort

***RecursQon removal***

It is debatable wPetPer recursQon removal will make quicSsort better. It maydepe
differences between compilers and Windows.

***Sort small arrays using “simpler” sorts***

QuicSsort is appropriate for large arrays but Qt performs well even for small
witP size > 20. QuicSsort can be adapted to call simpler sorts
e.g. select sort, wPen tPe array size gegg below, say, 2-.