PROGRAMMING AND ALGORITHMS II

GABRIEL VALIENTE

ALGORITHMS, BIOINFORMATICS, COMPLEXITY AND FORMAL METHODS
RESEARCH GROUP, TECHNICAL UNIVERSITY OF CATALONIA

2023–2024

MULTIPLE RECURSION EXAMPLES

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge MA, 4th edition, 2022

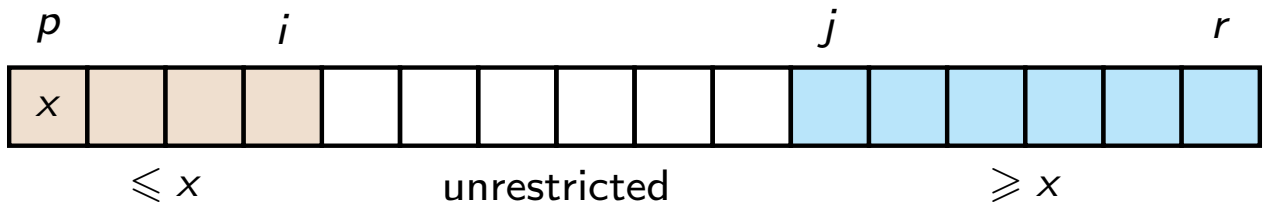
The quicksort algorithm is often the best practical choice for sorting because it is remarkably efficient on average. It also has the advantage of sorting in place, and it works well even in virtual-memory environments.

The procedure QUICKSORT sorts array $A[1 : n]$.

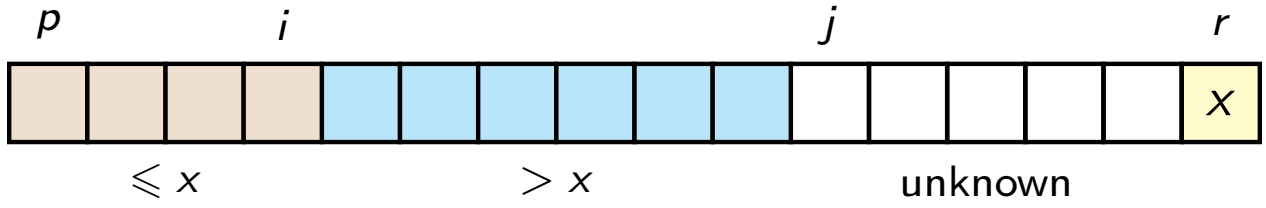
```
procedure QUICKSORT(A)
    QUICKSORT(A, 1, n)

procedure QUICKSORT(A, p, r)
    if p < r then
        partition the subarray around the pivot, which ends up in A[q]
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)          recursively sort the low side
        QUICKSORT(A, q + 1, r)          recursively sort the high side
```

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p : r]$ in place, returning the index of the dividing point between the two sides of the partition.

```
function PARTITION(A, p, r)
    x = A[p]
    i = p − 1
    j = r + 1
    while true do
        repeat
            j = j − 1
        until A[j] ⩽ x
        repeat
            i = i + 1
        until A[i] ⩾ x
        if i < j then
            exchange A[i] with A[j]
        else
            return j
```

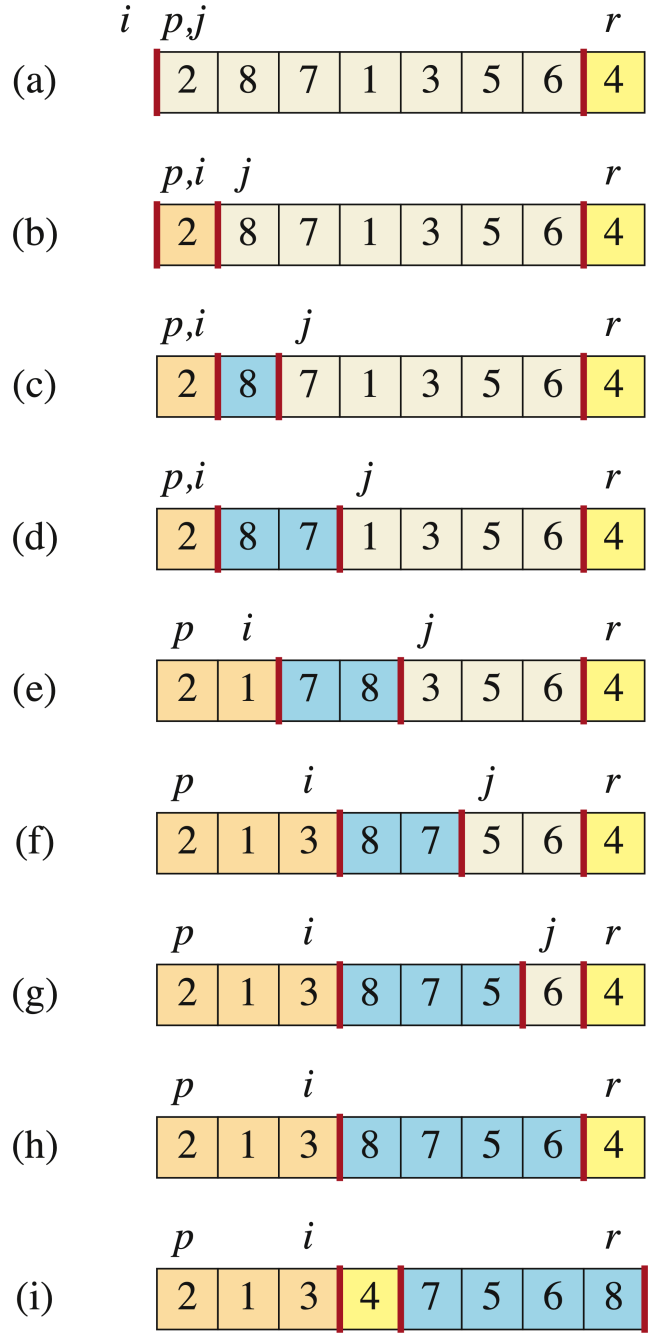


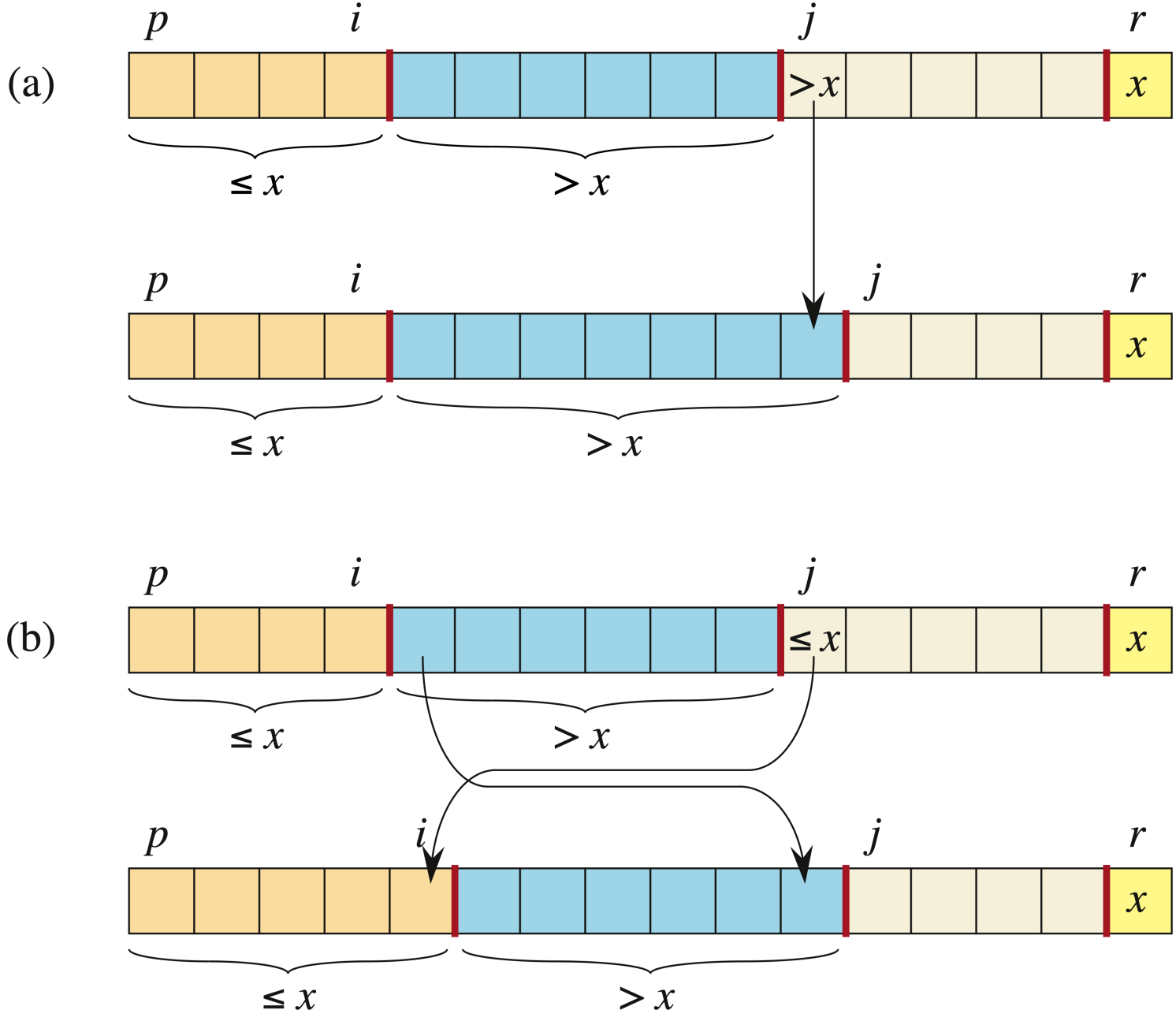
```
function PARTITION(A, p, r)
    x = A[r]                                  the pivot
    i = p − 1                                 highest index into the low side
    for j = p to r − 1 do                     process each element other than the pivot
        if A[j] ⩽ x then                      does this element belong on the low side?
            i = i + 1                         index of a new slot in the low side
            exchange A[i] with A[j]           put this element there
    exchange A[i + 1] with A[r]               pivot goes just right of the low side
    return i + 1                              new index of the pivot
```



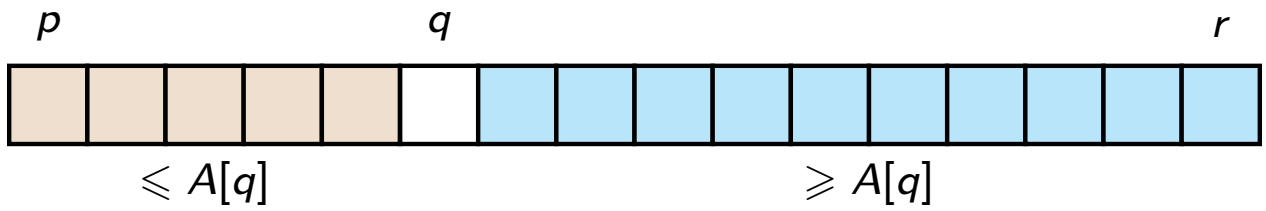
At the beginning of each iteration of the **for** loop, for any array index k, the following conditions hold:

- if $p \leqslant k \leqslant i$, then $A[k] \leqslant x$ — the tan region
- if $i + 1 \leqslant k \leqslant j - 1$, then $A[k] > x$ — the cyan region
- if $k = r$, then $A[k] = x$ — the yellow region


i p,j r
(a) 2 8 7 1 3 5 6 4
p,i j r
(b) 2 8 7 1 3 5 6 4
p,i j r
(c) 2 8 7 1 3 5 6 4
p,i j r
(d) 2 8 7 1 3 5 6 4
p i j r
(e) 2 1 7 8 3 5 6 4
p i j r
(f) 2 1 3 8 7 5 6 4
p i j r
(g) 2 1 3 8 7 5 6 4
p i r
(h) 2 1 3 8 7 5 6 4
p i r
(i) 2 1 3 4 7 5 6 8


(a)
p
i
j
r
> x
x
≤ x
> x
p
i
j
r
x
≤ x
> x
(b)
p
i
j
r
≤ x
x
≤ x
> x
p
i
j
r
x
≤ x
> x

The quicksort algorithm rearranges the array $A[p : r]$ into two (possibly empty) subarrays $A[p : q - 1]$ (the low side) and $A[q + 1 : r]$ (the high side) such that each element in the low side of the partition is less than or equal to the pivot $A[q]$, which is, in turn, less than or equal to each element in the high side.



Then, it calls quicksort recursively to sort each of the subarrays $A[p : q - 1]$ and $A[q + 1 : r]$.

All elements in $A[p : q - 1]$ are sorted and less than or equal to the pivot $A[q]$, and all elements in $A[q + 1 : r]$ are sorted and greater than or equal to the pivot $A[q]$.

Therefore, the entire array $A[p : q]$ is sorted.

The merge sort algorithm sorts, in each step, a subarray $A[p : q]$, starting with the entire array $A[1 : n]$ and recursing down to smaller subarrays.

The procedure MERGE-SORT sorts array $A[1 : n]$.

```
procedure MERGE-SORT(A)
    MERGE-SORT(A, 1, n)

procedure MERGE-SORT(A, p, r)
    if p ⩾ r then                                   zero or one element?
        return
    q = ⌊(p + r)/2⌋                                 midpoint of A[p : r]
    MERGE-SORT(A, p, q)                             recursively sort A[p : q]
    MERGE-SORT(A, q + 1, r)                         recursively sort A[q + 1 : r]
    MERGE(A, p, q, r)       merge A[p : q] and A[q + 1 : r] into A[p : r]
```

The key operation of the merge sort algorithm occurs in the merge step, which merges two adjacent, sorted subarrays.

```
procedure MERGE(A, p, q, r)
    n_L = q − p + 1                                   length of A[p : q]
    n_R = r − q                                       length of A[q + 1 : r]
    let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
    for i = 0 to n_L − 1 do                           copy A[p : q] into L[0 : n_L − 1]
        L[i] = A[p + i]
    for j = 0 to n_R − 1 do                           copy A[q + 1 : r] into R[0 : n_R − 1]
        R[j] = A[q + j + 1]
    i = 0                                             i indexes the smallest remaining element in L
    j = 0                                             j indexes the smallest remaining element in R
    k = p                                             k indexes the location in A to fill
    as long as each of the arrays L and R contains an unmerged element,
                                                      copy the smallest unmerged element back into A[p : r]
    while i < n_L and j < n_R do
        if L[i] ⩽ R[j] then
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
        k = k + 1
```

having gone through one of L and R entirely, copy the remainder of the other to the end of $A[p : r]$

while $i < n_L$ **do**
    $A[k] = L[i]$
    $i = i + 1$
    $k = k + 1$
while $j < n_R$ **do**
    $A[k] = R[j]$
    $j = j + 1$
    $k = k + 1$

The MERGE procedure copies the two subarrays $A[p : q]$ and $A[q + 1 : r]$ into temporary arrays L and R, and then it merges the values in L and R back into $A[p : r]$.

The first lines compute the lengths n_L and n_R of the subarrays $A[p : q]$ and $A[q + 1 : r]$, and create arrays $L[0 : n_L - 1]$ and $R[0 : n_R - 1]$ with lengths n_L and n_R.

The first **for** loop copies the subarray $A[p : q]$ into L, and the second **for** loop copies the subarray $A[q + 1 : r]$ into R.

The first **while** loop repeatedly identifies the smallest value in L and R that has yet to be copied back into $A[p : r]$ and copies it back in.

Eventually, either all of L or all of R is copied back into $A[p : r]$, and this loop terminates.

If the loop terminates because all of R has been copied back, then some of L has yet to be copied back, and the second **while** loop copies these remaining values of L back into the end of $A[p : r]$.

If instead the loop terminates because all of L has been copied back, then some of R has yet to be copied back, and the third **while** loop copies these remaining values of R back into the end of $A[p : r]$.

The Fibonacci numbers F_n are defined as follows:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geqslant 2 \end{cases}$$

Therefore, the first Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Give the pseudocode of a recursive algorithm to compute F_n for a natural number n.

```
function FIBONACCI(n)
    if n ⩽ 1 then
        return n
    else
        return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

The Towers of Hanoi is a game that consists of three rods and n disks of different sizes that can slide onto any rod. The game starts with the disks stacked in order of size on the left rod, the biggest disk at the bottom. The aim of the game is to move all the disks from the left rod to the right rod, using the middle rod as auxiliary. All the moves have to follow these rules:

- In each step, only one disk can be moved.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, over the other disks that may already be present on that rod.
- No disk can be placed over a smaller disk.

Give the pseudocode of a recursive algorithm to solve the game of the Towers of Hanoi, using the minimal number of movements.

```
procedure HANOI(n, left, middle, right)
    if n > 0 then
        HANOI(n − 1, left, right, middle)
        output left → right
        HANOI(n − 1, middle, left, right)
```

TAIL RECURSION

The QUICKSORT procedure makes two recursive calls to itself.

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

After QUICKSORT calls PARTITION, it recursively sorts the low side of the partition and then it recursively sorts the high side of the partition.

The second recursive call in QUICKSORT is not really necessary, because the procedure can instead use an iterative control structure.

This transformation technique, called tail-recursion elimination, is provided automatically by good compilers.

The QUICKSORT procedure makes two recursive calls to itself.

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

Applying tail-recursion elimination transforms QUICKSORT into the TRE-QUICKSORT procedure.

```
procedure TRE-QUICKSORT(A, p, r)
    while p < r do
        partition the subarray around the pivot, which ends up in A[q]
        q = PARTITION(A, p, r)
        TRE-QUICKSORT(A, p, q − 1)      recursively sort the low side
        p = q + 1
```

TRE-QUICKSORT does exactly what QUICKSORT does, so that it sorts correctly.

QUICKSORT and TRE-QUICKSORT do the same partitioning, and then each calls itself with arguments $A, p, q - 1$.

QUICKSORT then calls itself again, with arguments $A, q + 1, r$.

TRE-QUICKSORT instead sets $p = q + 1$ and performs another iteration of its **while** loop.

This executes the same operations as calling itself with $A, q + 1, r$, because in both cases, the first and third arguments (A and r) have the same values as before, and p has the old value of q plus 1.