ALGORITHMS AND DATA STRUCTURES I

Course 8

Current Course

The basic idea of the division technique

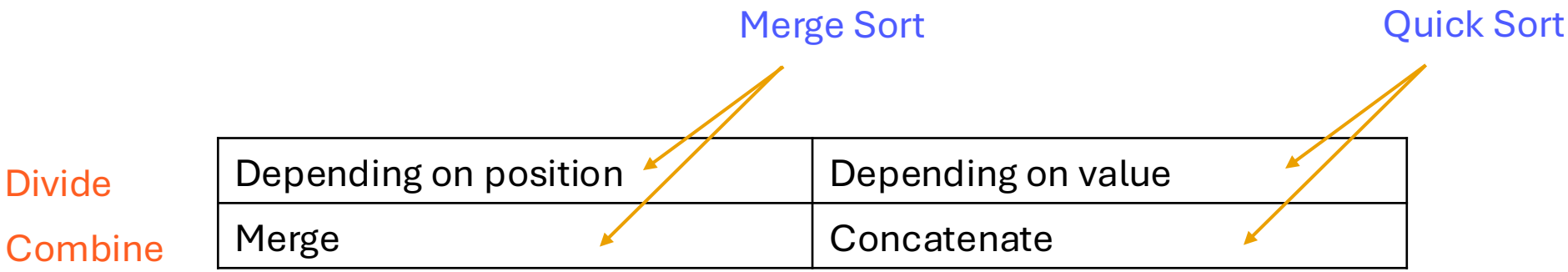
Examples

Master theorem for estimating the order of complexity of algorithms based on reduction/division techniques

Sort by MergeSort

Efficient sorting

- Elementary sorting methods belong to $O(n^2)$
- Idea to streamline the sorting process:
  - Divide the initial sequence into two subsequences
  - Sort each subsequence
  - Combine sorted subsequences



Quick sort

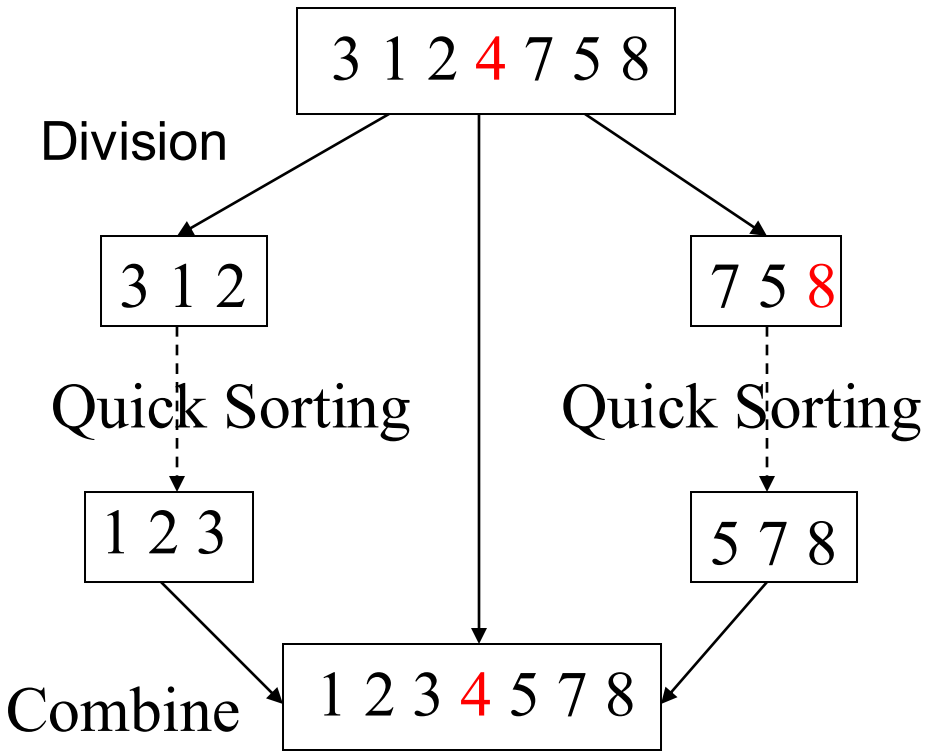
Idea:

- Reorganize and divide x[1..n] into two subarrays x[1..q] and x[q+1..n] so that the elements of x[1..q] are smaller than x[q+1..n]
- Sort each of the subarrays applying the same strategy
- Concatenate sorted subarrays

- Creator: Tony Hoare (1962)

Quick sort

Example 1



- An x[q] element having the properties:

(a) x[q]>=x[i], for all i<q

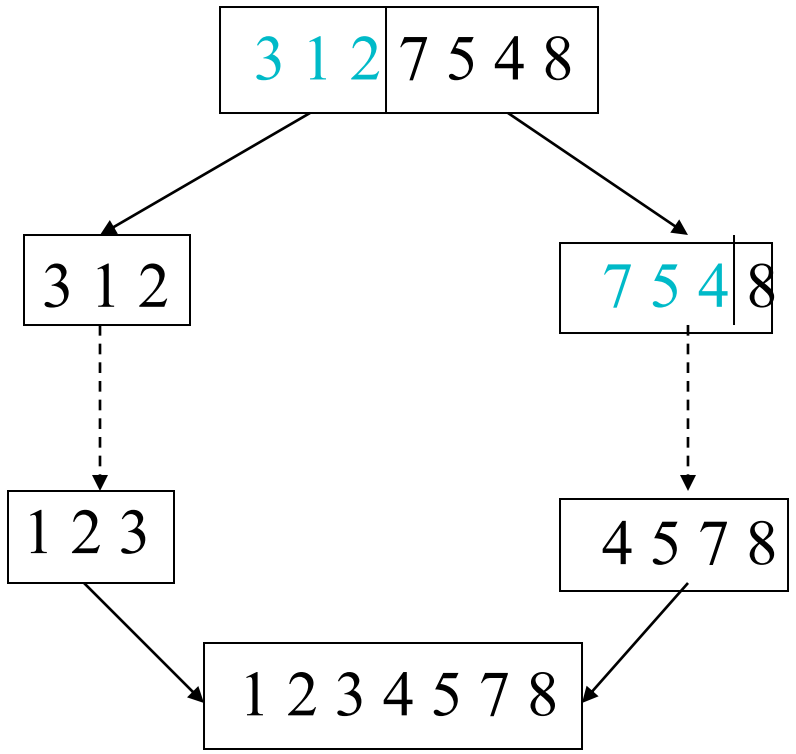
(b) x[q]<=x[i], for all i>q

It is referred to as pivot

- A pivot is an element in its final position
- A good pivot divides the current array into two subarrays of close size (balanced partitioning)
- Sometimes the pivot divides the array unevenly
- Other times there is no such pivot (e.g. (3 1 2)). In this case, a pivot must be created by interchanging elements

Quick sort

Example 2



- A position q having the property:

  (a) x[i]<=x[j], for 1<=i<=q and q+1<=j<=n

  It is referred to as the partitioning position
- A good partitioning position divides the current array into subarrays of close size
- Sometimes the partitioning position divides the board unevenly
- Other times there is no partitioning position. In this case, such a position is created by interchanging some elements

Quick sort

Variant using pivot

```
quicksort1(x[s..d])
IF s<d THEN
  q ← pivot(x[s..d])
  x[s..q-1] ← quicksort1(x[s..q-1])
  x[q+1..d] ← quicksort1(x[q+1..d])
ENDIF
RETURN x[s..d]
```

Variant using partitioning position

```
quicksort2(x[s..d])
IF s<d THEN
  q ← partitie(x[s..d])
  x[s..q] ← quicksort2(x[s..q])
  x[q+1..d] ← quicksort2(x[q+1..d])
ENDIF
RETURN x[s..d]
```

Quick sort

Building a pivot:

- Choose an arbitrary value from the array (first, last, or random) – this will represent the pivot value

- Rearrange the elements of the array so that all items that are smaller than the chosen value are in the first part of the array and values greater than the pivot are in the second part of the board

- Place the pivot value at its final position (so that all elements to its left are smaller and all elements to its right are larger)

Quick sort

An idea for rearranging elements:

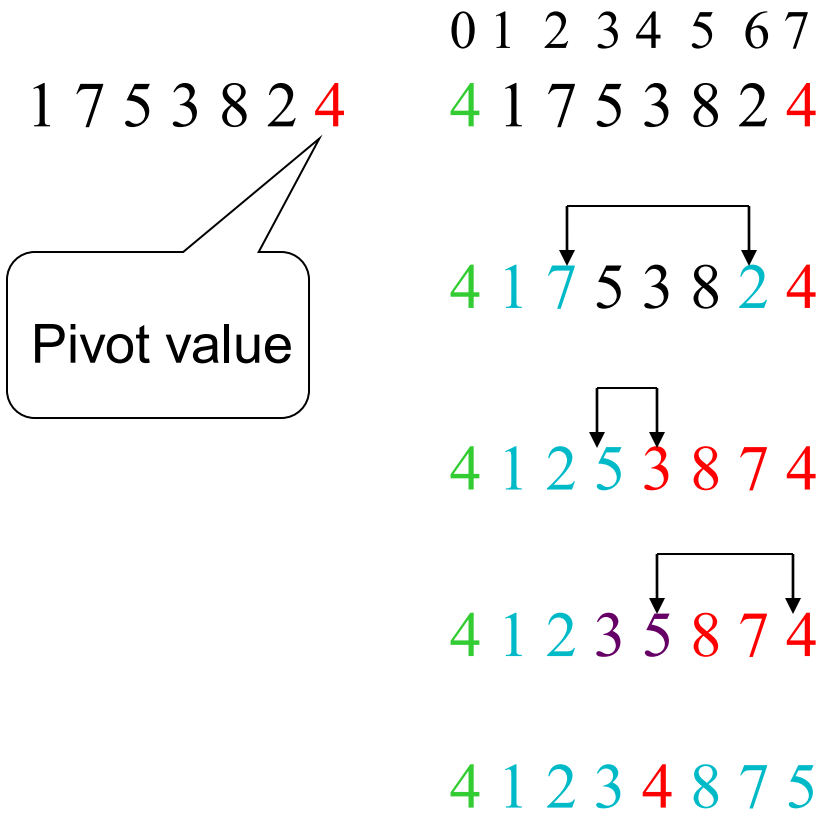
- Two indicators are used: one starting from the first element and the other starting from the last element
- The indicators are increased or decreased until an inversion is identified.

Inversion: pair of i<j indices with property that x[i]>pivot and x[j]<pivot

- Fix inversion by swapping items
- Continue the process until the indicators "intersect"

Quick sort

Building a pivot



- Choose pivot value: 4 (last value in array)
- Place a sentinel before the first position of the array (for the original array only)

i=0, j=7

i=2, j=6

i=3, j=4

i=4, j=3 (indicators crossed)

The pivot is placed on its final position

Quick sort

```
pivot(x[s..d])
   v ← x[d]
   i ← s-1
   j ← d
   WHILE i<j DO
      REPEAT i ← i+1 UNTIL x[i]>=v
      REPEAT j ← j-1 UNTIL x[j]<=v
      IF i<j THEN x[i]↔x[j] ENDIF
  ENDWHILE
  x[i] ↔ x[d]
  RETURN i
```

Remarks:

- x[d] plays the role of a sentinel at the far right
- At the far left, a sentinel value x[0]=v is explicitly placed (only for the initial array x[1..n])
- Conditions x[i]>=v, x[j]<=v allow search to stop when sentinels are encountered. They also allow to obtain a balanced partitioning when the board contains identical elements.
- At the end of the cycle, while indicators satisfy either i=j or i=j+1

Quick sort

```
pivot(x[s..d])
   v ← x[d]
   i ← s-1
   j ← d
   WHILE i<j DO
      REPEAT i ← i+1 UNTIL x[i]>=v
      REPEAT j ← j-1 UNTIL x[j]<=v
      IF i<j THEN x[i]↔x[j] ENDIF
  ENDWHILE
  x[i] ↔ x[d]
  RETURN i
```

Correctness:

Invariant:

If i<j then x[k]<=v for k=s.. i

x[k]>=v for k=j.. d

If i>=j then x[k]<=v for k=s.. i

x[k]>=v for k=j+1..d

Quick sort

```
pivot(x[s..d])
   v ← x[d]
   i ← s-1
   j ← d
   WHILE i<j DO
      REPEAT i ← i+1 UNTIL x[i]>=v
      REPEAT j ← j-1 UNTIL x[j]<=v
      IF i<j THEN x[i]↔x[j] ENDIF
  ENDWHILE
  x[i] ↔ x[d]
  RETURN i
```

Efficiency:

Pb size: n=d-s+1

Op. dominant: comparison in which elements of x intervene

T(n)=n+c,

c=0 if i=j

and

c=1 if i=j+1

So T(n) belongs to Θ (n)

Quick sort

Remark: pivot position doesn't always divide the array evenly

Balanced partitioning:

- The array is divided into two sub-arrays of size close to n/2
- If each partitioning is balanced, then the algorithm performs fewer operations (corresponds to the most favorable case);

Unbalanced partitioning:

- The array is divided into a subarray with (n-1) elements, the pivot, and an empty subarray
- If each partitioning is unbalanced, then the algorithm performs several operations (corresponds to the worst case);

Quick sort

Worst case analysis:

$$T(n)=\begin{cases}0 & \text{if } n=1\\ T(n-1)+n+1, & \text{if } n>1\end{cases}$$

Reverse substitution:

T(n)=T(n-1)+(n+1)

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

…

T(2)=T(1)+3

T(1)=0

T(n)=(n+1)(n+2)/2-3

In the worst case, the algorithm is of quadratic complexity

So quick sorting belongs to $O(n^2)$

Quick sort

Best case analysis:

$$T(n)=\begin{cases}0 & \text{if } n=1\\ 2T(n/2), & \text{if } n>1\end{cases}$$

Applying the second case of the "master" theorem (for k=2,m=2,d=1) it follows that in the most favorable case the order of complexity is nlog(n)

So the quick sort algorithm belongs to $\Omega(n\log(n))$ and $O(n^2)$

Analysis in the medium case could be useful

Quick sort

Analysis in the average case.

Assumptions:

- Each partitioning step requires at most (n+1) comparisons
- There are n possible positions for the pivot. We assume that each of these positions has the same chance of being selected (Prob(q)=1/n)
- If the pivot is in position q, then the number of comparisons satisfies

$$T_q(n)=T(q-1)+T(n-q)+(n+1)$$

Quick sort

The average number of comparisons is

$T_a(n)=(T_1(n)+\ldots+T_n(n))/n$

$=((T_a(0)+T_a(n-1))+(T_a(1)+T_a(n-2))+\ldots+(T_a(n-1)+T_a(0)))/n + (n+1)$

$=2(T_a(0)+T_a(1)+\ldots+T_a(n-1))/n+(n+1)$

Deci

n $T_a(n) = 2(T_a(0)+T_a(1)+\ldots+T_a(n-1))+n(n+1)$

$(n-1)T_a(n-1)= 2(T_a(0)+T_a(1)+\ldots+T_a(n-2))+(n-1)n$

Computing the difference between the last two equalities: $nT_a(n)=(n+1)T_a(n-1)+2n$

$T_a(n)=(n+1)/n\ T_a(n-1)+2$

Quick sort

Analysis in the average case.

By reverse substitution:

$T_a(n) = (n+1)/n \; T_a(n-1)+2$

$T_a(n-1)= n/(n-1) \; T_a(n-2)+2 \quad |*(n+1)/n$

$T_a(n-2)= (n-1)/(n-2) \; T_a(n-3)+2 \; |*(n+1)/(n-1)$

…

$T_a(2) = 3/2 \; T_a(1)+2 \quad |*(n+1)/3$

$T_a(1) = 0 \quad |*(n+1)/2$

$T_a(n) = 2+2(n+1)(1/n+1/(n-1)+…+1/3) \approx 2(n+1)(\ln n-\ln 3)+2$

In the average case, the order of complexity is nlog(n)

Quick sorting -variants

Another variant of pivot construction

```
pivot(x[s..d])
   v ← x[s]
   i ← s
   FOR j ← s+1,d
      IF x[j]<=v THEN
         i ← i+1
         x[i] ↔ x[j]
      ENDIF
   ENDFOR
x[s] ↔x[i]
RETURN i
```

Invariant: x[k]<=v pentru s<=k<=i
x[k]>v pentru i<k<=j-1

3 7 5 2 1 4 8 v=3, i=1,j=2

3 7 5 2 1 4 8 i=2, j=4

3 2 5 7 1 4 8 i=3, j=5

3 2 1 7 5 4 8 i=3, j=8

Pivot Placement:
1 2 3 7 5 4 8

Pivot position: 3
Pivot build complexity order: O(n)

Quick sorting -variants

Build a partitioning position

```
Partiție(x[s..d])
   v ← x[s]
   i ← s-1
   j ← d+1
   WHILE i<j DO
      REPEAT i ← i+1 UNTIL x[i]>=v
      REPEAT j ← j-1 UNTIL  x[j]<=v
      IF i<j THEN x[i] ↔x[j]
      ENDIF
  ENDWHILE
RETURN j
```

3 7 5 2 1 4 8	v=3
3 7 5 2 1 4 8	i=2, j=5
3 1 5 2 7 4 8	i=3, j=4
3 1 2 5 7 4 8	i=4, j=3

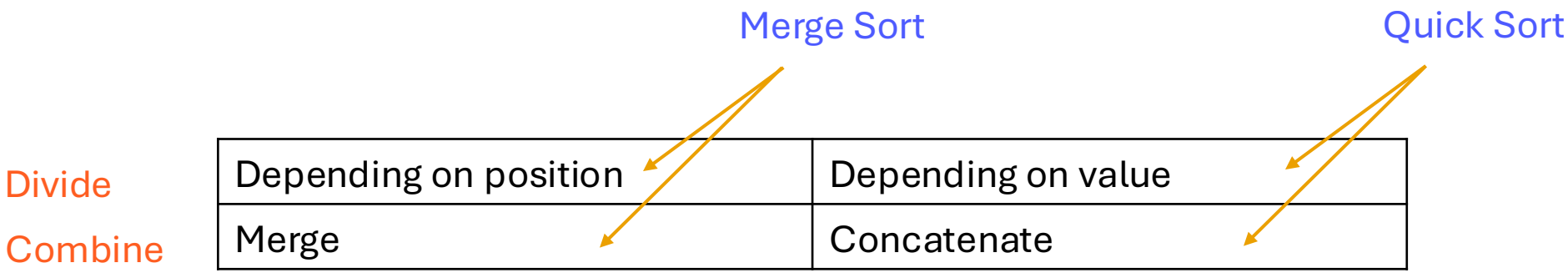
Partitioning position: 3
Complexity order: O(n)

Note: The partitioning algorithm is used in the quicksort2

Efficient sorting

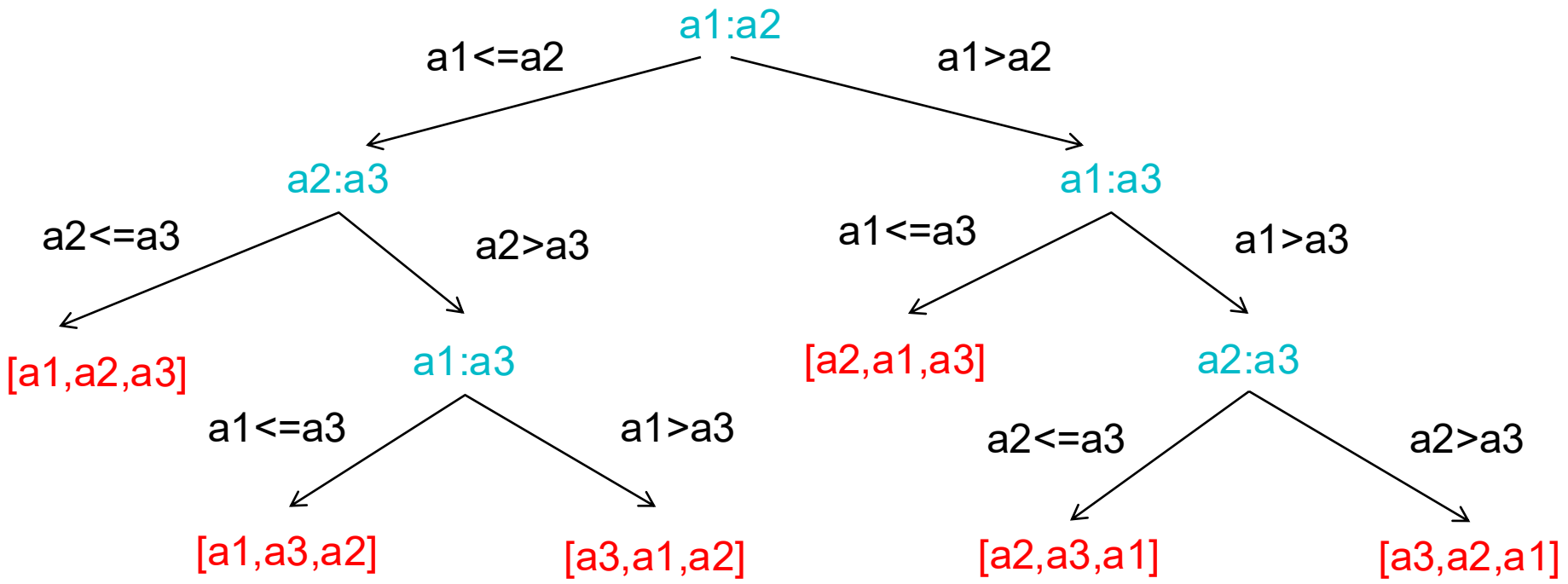
MergeSort – O(n log(n))

QuickSort - O(n log(n))



Complexity of sorting based on comparisons

Example of a binary decision tree (n=3, [a1,a2,a3]):



Remak: each of the n! variants of rearranging the list of values must appear in at least one leaf of the tree

Complexity of sorting based on comparisons

Remark:

Each of the n! variants of rearranging the list of values must appear in at least one leaf of the tree

The sorting process for a given list corresponds to going through a branch in the tree from the root to a leaf node

The number of worst-case comparisons is correlated with the length of the longest branch in the tree = tree height = h

The maximum number of leaves of a binary tree of height h is 2^h

So $n! <= \text{nr leafs} <= 2^h$

Complexity of sorting based on comparisons

So

$n! <= \text{nr leafs} <= 2^h$

In other words $\log n! <= h$

Using approximation (Stirling's formula)

$$n! \cong \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

It follows that $h >= \log n! = \Theta(n \log n)$

Other applications of division: selection of the k-th element

Given an x[1..n] (unordered) array, one considers the problem of determining the k-th element in ascending order

Example: x=[4,1,5,3,8,6]

k=1=> 1 (smallest element)

k=3=> 4

K=6=> 8 (largest element)

Variants of solution:

Sort x[1..n] => O(nlogn)

Partial sort by selection => O(kn) – effective only for small k (or close to n)

Is there a more efficient option?

Other applications of division: selection of the k-th element

Idea: the partitioning strategy from quicksort is used and, depending on the relationship between the current value of k and the number of elements in x[s.. q] continue the search in the first part (x[s.. q]) or in the second part (x[q+1..d])

Remark: if for the initial call k is between 1 and n, for each call, k will be between 1 and d-s+1 (k is the rank of an item but not the index of the element)

```
selection(x[s..d],k)
  if s==d then return x[s]
  else
     q←partitie(x[s..d])
     r ← q-s+1
     if k<=r then return selection(x[s..q],k)
                 else return selection(x[q+1..d],k-r)
     endif
  endif
```

Other applications of division: selection of the k-th element

```
selection(x[s..d],k)
  if s==d then return x[s]
  else
     q←partitie(x[s..d])
     r ← q-s+1
     if k<=r then return selection(x[s..q],k)
                 else return selection(x[q+1..d],k-r)
     endif
  endif
```

Most favourable case (balanced partitioning):

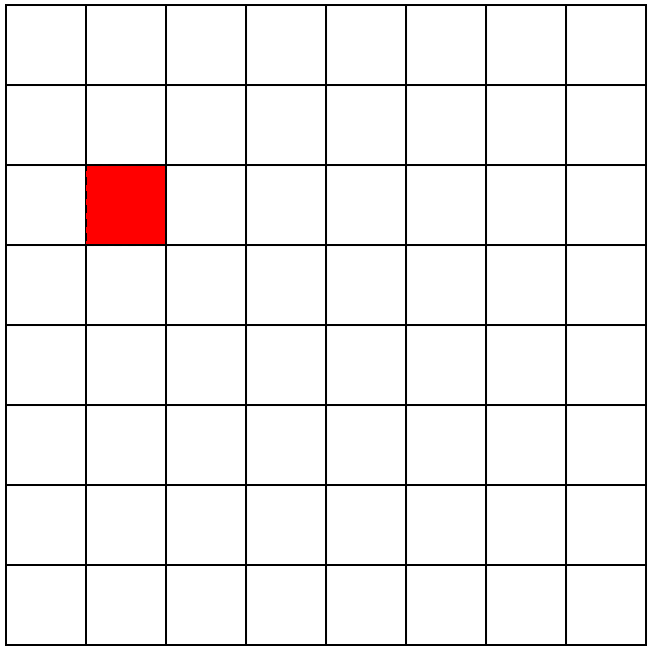
$$T(n)=\begin{cases}0 & n=1\\ T(n/2)+n & n>1\end{cases}$$

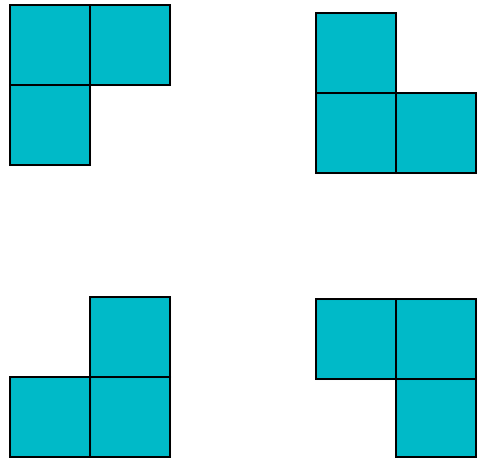
$\Rightarrow$(t. Master, caz 1: m=2,k=1,d=1)

T(n) belongs to O(n)

Other applications of division: Triomino

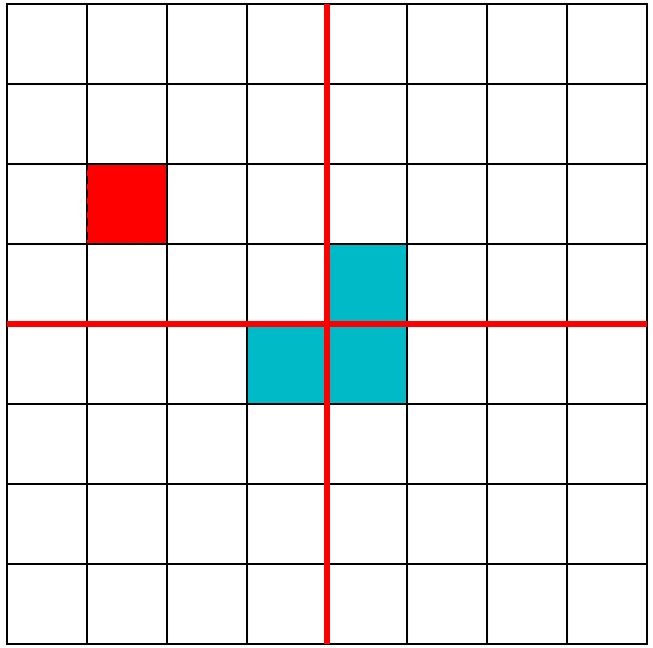
Consider a square grid of side n=2k in which one of the cells is marked (forbidden). There is the problem of covering the grid with pieces consisting of 3 cells placed in the shape of L (four variants that can be obtained by rotating by 90 degrees each)

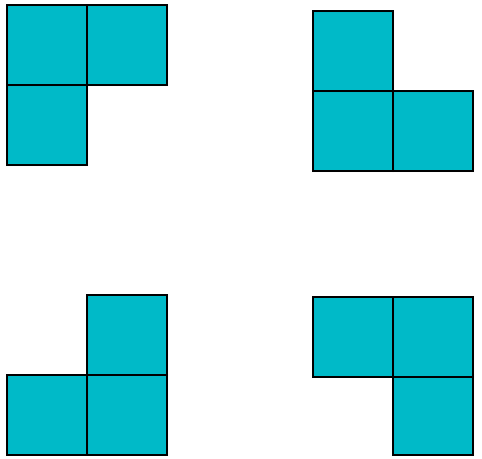




Other applications of division: Triomino

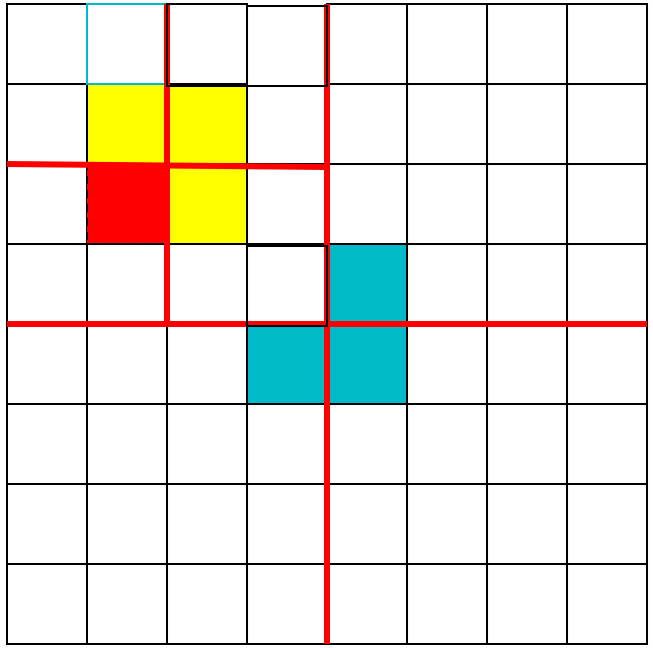
Idea: it reduces down to 4 similar size issues 2^k by placing a piece in the central area so that it occupies a cell in each of the 3 areas that have all the cells free

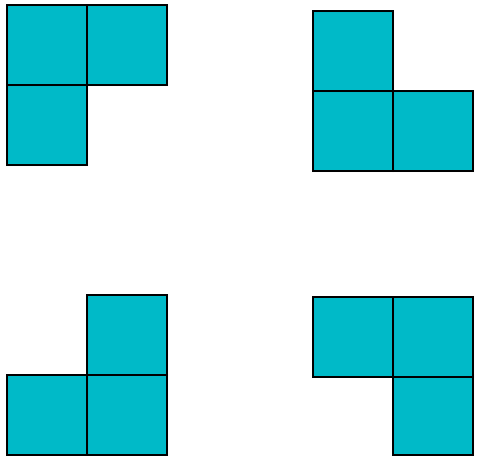




Other applications of division: Triomino

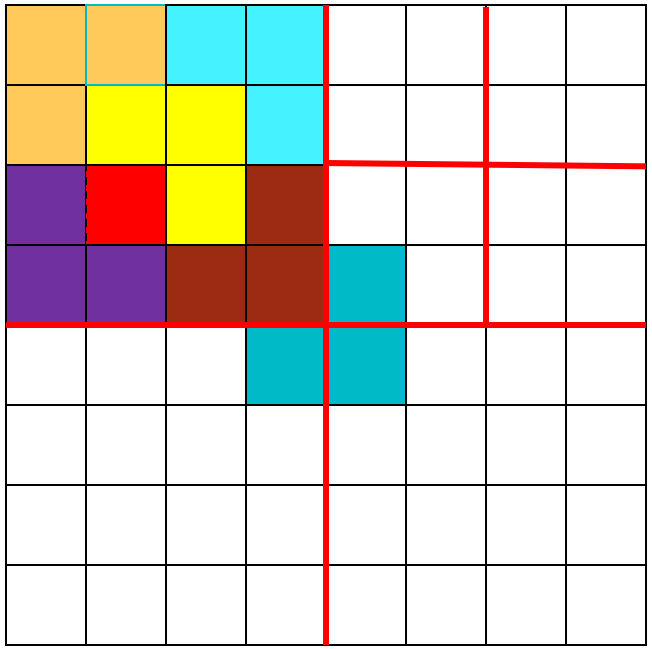
Idea: the same strategy applies to the upper left area

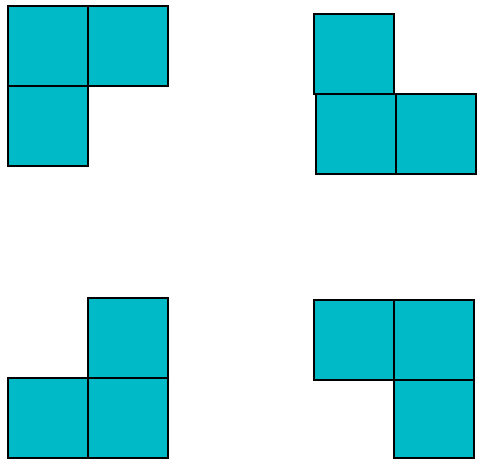




Other applications of division: Triomino

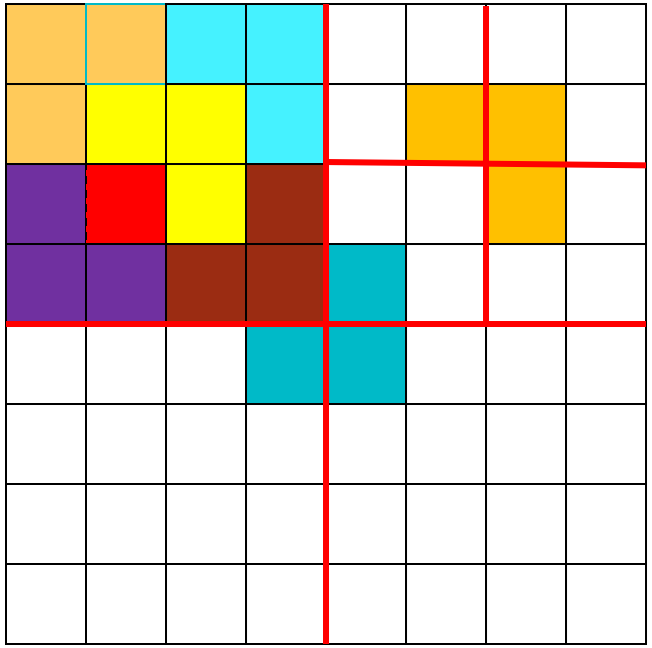
Idea:... The same strategy applies to the upper right area

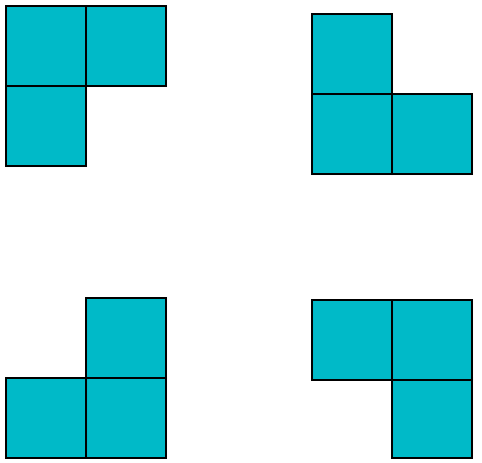




Other applications of division: Triomino

Idea:... The same strategy applies to the upper right area

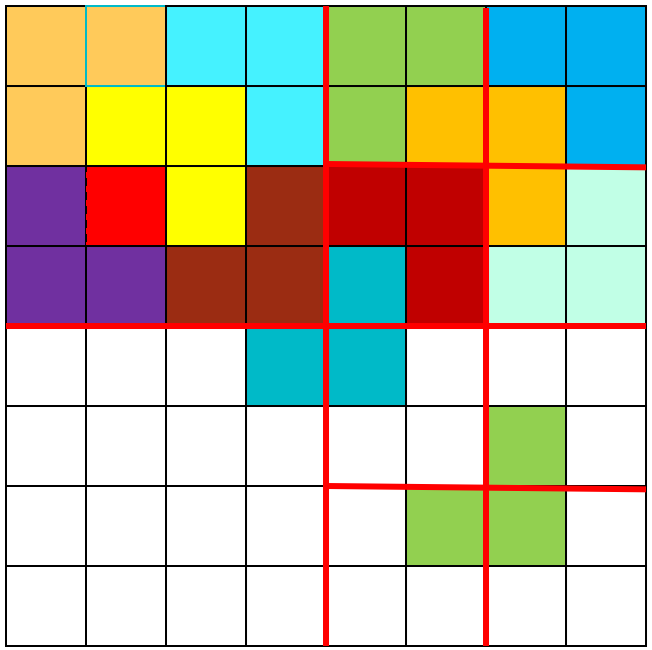


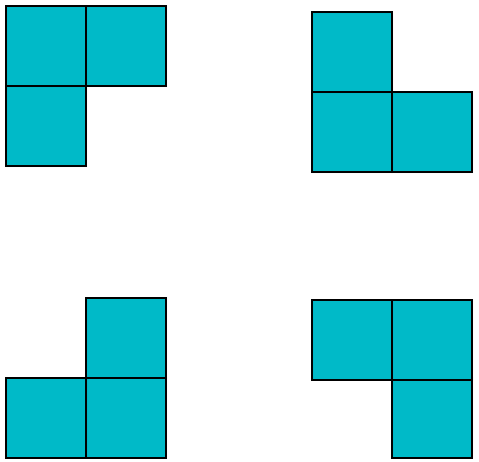


Other applications of division: Triomino

Idea: then for the lower right area and finally for the bottom left area

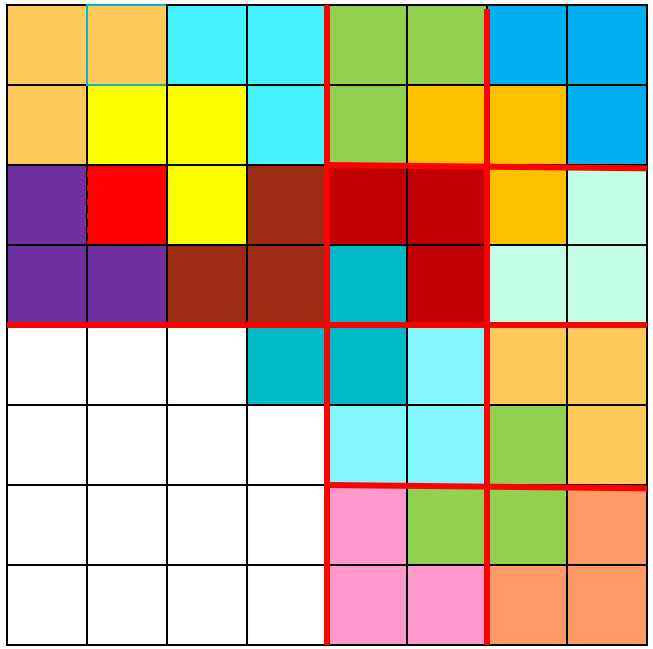
Note: it does not matter the order in which subproblems are solved

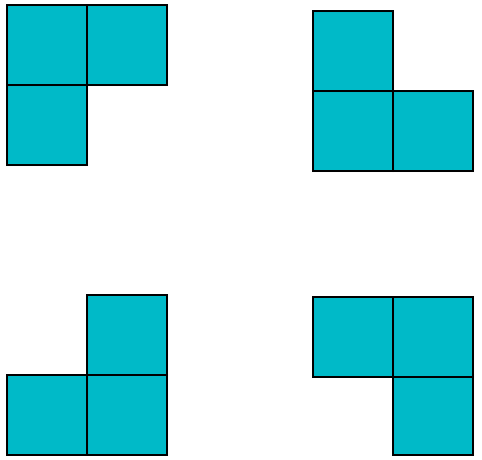




Other applications of division: Triomino

Idea: then for the bottom right area

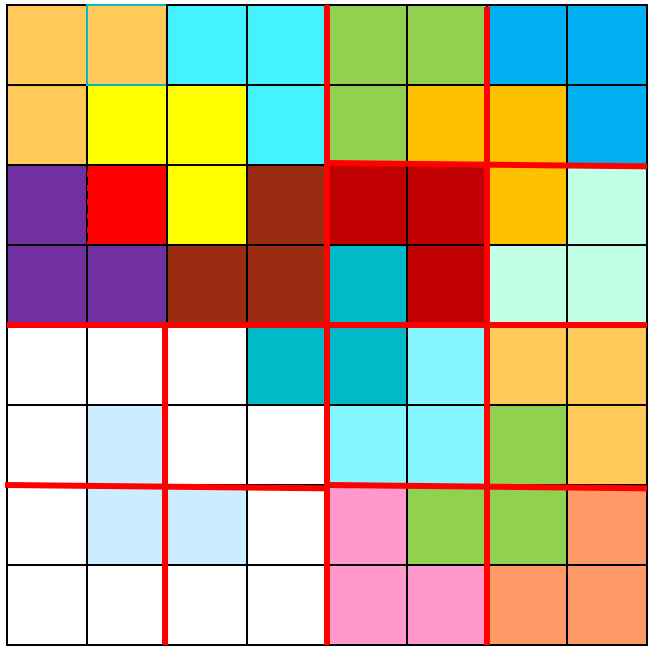


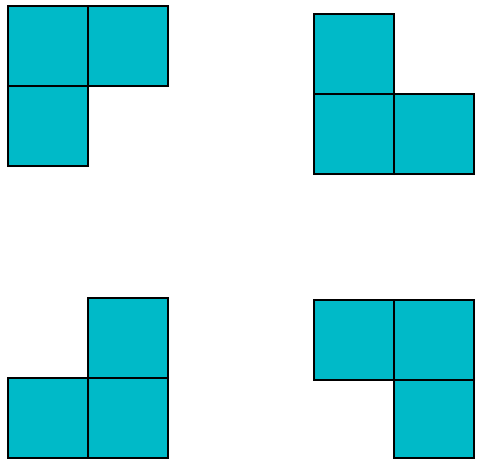


Other applications of division: Triomino

Idea:... and finally for the lower left area

Note: it does not matter the order in which subproblems are solved





Other applications of division: Triomino

Algoritm Ideas:

```
nr ← 0; // nr ordine piesa
Triomino(i1,j1,i2,j2,ih,jh) // i1,j1,i2,j2=indici colturi grila,  ih,jh=indici celula ocupata
 if ((i2-i1==1) and (j2-j1==1))  then <completare cele 3 celule libere>
 else
    imij ←(i1+i2)/2;  jmij ←(j1+j2)/2  // calcul indici mijloc
    if (ih<=imij) and (jh<=jmij) then    // celula ocupata e in subgrila stanga sus
         a[imij][jmij+1] ← nr; a[imij+1][jmij] ← nr; a[imij+1][jmij+1] ← nr;  nr=nr+1;
          triomino(i1,j1,imij,jmij,ih,jh);                    // subgrila stanga jos
          triomino(i1,jmij+1,imij,j2,imij,jmij+1);          // subgrila dreapta sus
          triomino(imij+1,jmij+1,i2,j2,imij+1,jmij+1);  // subgrila dreapta jos
          triomino(imij+1,j1,i2,jmij,imij+1,jmij);         // subgrila stanga jos
   if ((ih<=imij) and (jh>jmij)) then ….  // subgrila dreapta sus
   if ((ih>imij) and (jh>jmij)) then ….    // subgrila dreapta jos
   if ((ih>imij) and (jh<=jmij)) then ….  // subgrila stanga jos
```

The next course will be about...

... The technique of optimal local search

... and apps

Q&A