ALGORITHMS AND DATA STRUCTURES I

Course 8

Previous Course

... How to analyze the effectiveness of recursive algorithms

- Write the recurrence relationship corresponding to the execution time
- The recurrence relationship is solved using the direct or reverse substitution technique

... How to solve problems using the reduction technique

- Decrease by reducing problem size by a constant/variable
- Decrease by dividing the problem size by a constant/variable factor

... Sometimes the reduction technique leads to more efficient algorithms than those obtained by applying the brute force technique

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

MergeSort

The basic idea of the division technique

- The current problem is divided into several subproblems of the same type but of smaller size
  - The component subproblems must be independent (each of these subproblems will be solved at most once)
  - Subproblems must be close in size
- Subproblems are solved by applying the same strategy (algorithms designed using the division technique can be easily described in a recursive manner)
  - If the size of the problem is less than a certain value (critical size), then the problem is solved directly, otherwise it is solved by applying the division technique again (for example, recursively)
- If necessary, the solutions obtained by solving the subproblems are combined

The basic idea of the division technique

Divide&conquer (n)

IF $n<=n_c$ THEN r ← <solve P(n) directly to get the result r>

ELSE

<decompose P(n) in $P(n_1)$, ..., $P(n_k)$>

FOR i←1,k DO

r_i ← Divide&conquer(n_i) // solves subproblem P (n_i)

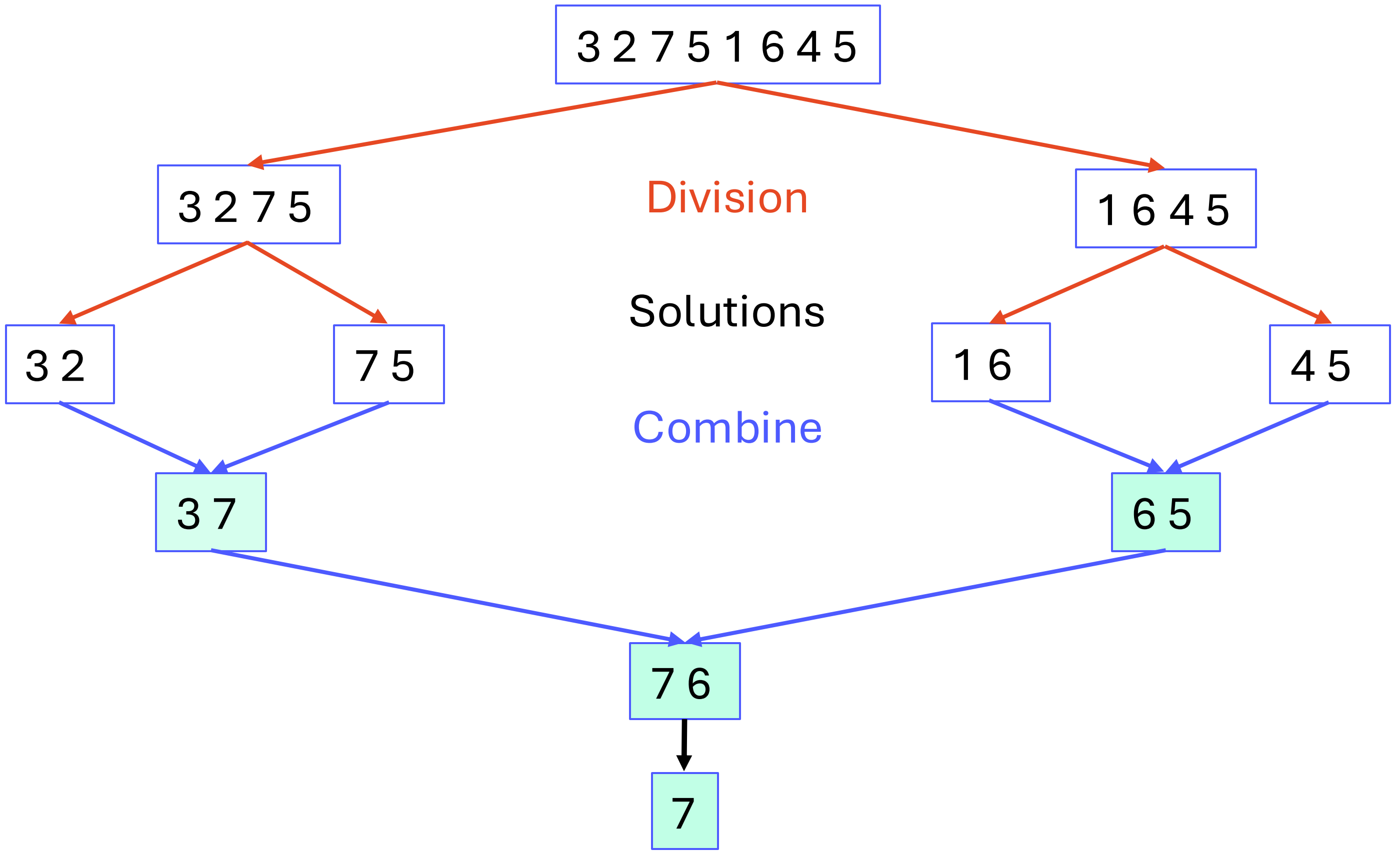
ENDFOR

r ← combine (r_1, ..., r_k)

ENDIF

RETURN r

Example 1
Maximum value of an array



Example 1
Maximum value of an array

Algorithm

```
maxim(x[s..d])
 IF s==d then RETURN x[s]
  ELSE
   m ←(s+d) DIV 2 //divide
   max1 ← maxim(x[s..m]) // solve
   max2 ← maxim(x[m+1..d]) //solve
   IF max1>max2 // combine
   THEN RETURN max1
   ELSE RETURN max2
   ENDIF
 ENDIF
```

Efficiency analysis

Problem size: n

Dominant operation: comparison

Recurrence relationship:

$$T(n) = \begin{cases} 0, & n = 1 \\ T\left(\left[\frac{n}{2}\right]\right) + T\left(n - \left[\frac{n}{2}\right]\right) + 1 & n > 1 \end{cases}$$

Example 1
Maximum value of an array

$$T(n) = \begin{cases} 0, & n = 1 \\ T\left(\left[\frac{n}{2}\right]\right) + T\left(n - \left[\frac{n}{2}\right]\right) + 1 & n > 1 \end{cases}$$

Particular case: $n=2^m$

$$T(n) = \begin{cases} 0, & n = 1 \\ 2T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

Reverse substitution

$T(2^m) = 2T(2^{m-1})+1$

$T(2^{m-1})=2T(2^{m-2})+1$ |* 2

...

$T(2)=2T(1)+1$ |* 2m-1

$T(1)=0$

$T(n)=1+\ldots+2^{m-1}=2^m-1=n-1$

Example 1
Maximum value of an array

$$T(n) = \begin{cases} 0, & n = 1 \\ T\left(\left[\frac{n}{2}\right]\right) + T\left(n - \left[\frac{n}{2}\right]\right) + 1 & n > 1 \end{cases}$$

Particularly case: $n=2^m$ => T(n)=n-1

General case

(a) Prove by mathematical induction

Check

n=1 =>T(n)=0=n-1

Induction step

Assume that T(k)=k-1 for any k<n.

Then T(n)=[n/2]-1+n-[n/2]-1+1=n-1

So T(n) =n-1 => T(n) belongs to Θ(n).

Example 1
Maximum value of an array

General case.

(b) The rule of "smooth" functions

If T(n) belongs to Θ(f(n)) for $n=b^m$

T(n) is increasing for large values of n

f(n) is "smooth" (f(cn) belongs to Θ(f(n)) for any positive constant c)

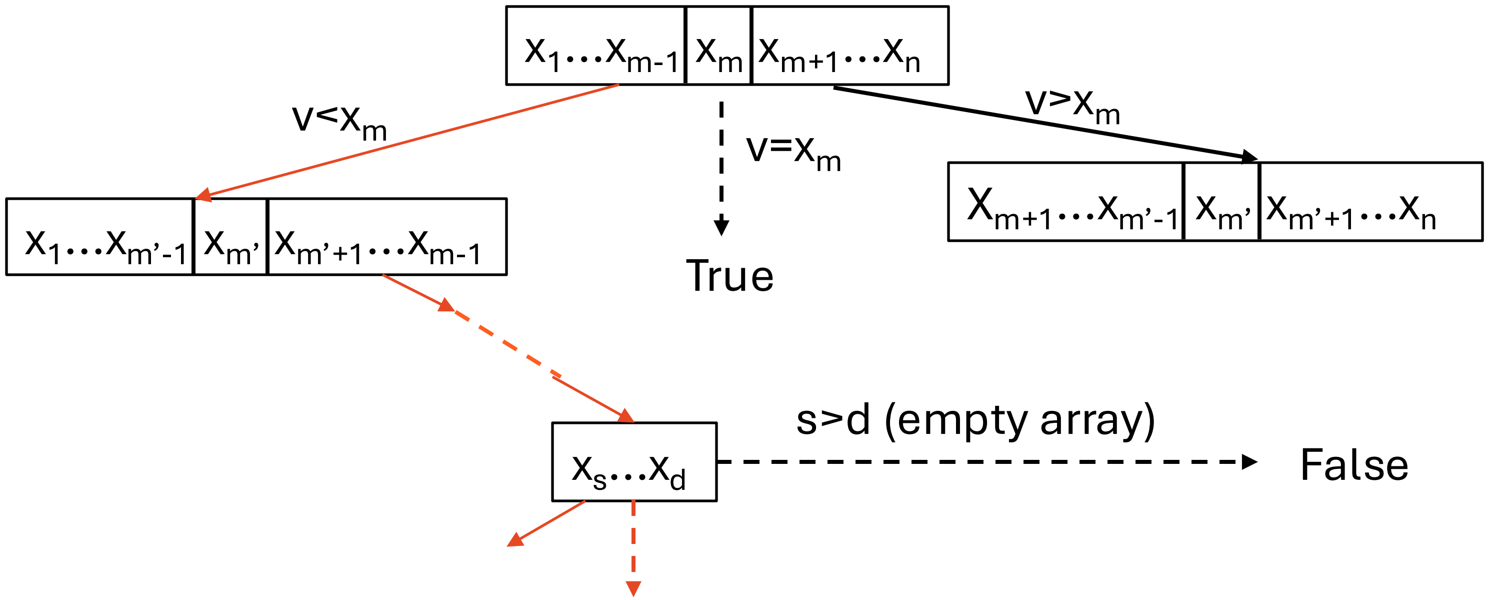
then T(n) belongs to Θ(f(n)) for any n

Remarks

- All functions that do not grow very rapidly (e.g. logarithmic and polynomial functions) are smooth functions. In contrast, the exponential function does not have this property: a^{cn} does not belong to $\Theta(a^n)$
- For the "maximum" algorithm: T(n) is ascending, f(n)=n is smooth, so T(n) is from Θ(n) for any value of n

Example 2
Binary search

- Check whether or not a given value, v, belongs to an ascending ordered array, x[1..n] (x[i]<=x[i+1], i=1..(n-1))
- Idea: Compare v to the element at the middle index. If v is equal, return the index. If v < middle element, search the left half; otherwise, search the right half.



Example 2
Binary search

Recursive variant

```
binsearch(x[s..d],v)
 IF s>d THEN RETURN False //base case empty array
 ELSE
   m ←(s+d) DIV 2 //start division step
  IF v==x[m] THEN RETURN True
  ELSE //recursive call
        IF v<x[m]  THEN RETURN binsearch(x[s..m-1],v)
                  ELSE RETURN binsearch(x[m+1..d],v)
        ENDIF
  ENDIF
 ENDIF
```

Function call

binsearch(x[1..n],v)

(first s=1, d=n)

Remark

$n_c=0$

k=2

Only one of the subproblems is solved

Binary search is actually based on the reduction technique (only one of the subproblems is solved)

Example 2
Binary search

Iterative variant 1

```
BinarySearch1(x[1..n],v)
 s ← 1
 d ← n
 WHILE s<=d DO
   m ←(s+d) DIV 2
   IF v==x[m] THEN RETURN True
   ELSE
     IF v<x[m]
     THEN d ← m-1
     ELSE s ← m+1
 ENDIF / ENDIF/ ENDWHILE
 RETURN False
```

Iterative variant 2

```
BinarySearch2(x[1..n],v)
 s ← 1
 d ← n
 WHILE s<d DO
   m ←(s+d) DIV 2
   IF v<=x[m]
   THEN d ← m
   ELSE s ← m+1
 ENDIF / ENDWHILE
 IF x[s]==v THEN RETURN True
            ELSE RETURN False
 ENDIF
```

Example 2
Binary search

Iterative variant 2

```
BinarySearch2(x[1..n],v)
 s ← 1
 d ← n
 WHILE s<d DO
   m ←(s+d) DIV 2
   IF v<=x[m]
   THEN d ← m
   ELSE s ← m+1
 ENDIF / ENDWHILE
 IF x[s]==v THEN RETURN True
            ELSE RETURN False
 ENDIF
```

Corectness

Precondition: n>=1

Postcondition: "returns True if v is in x[1..n] and False otherwise"

Invariant property: "v is in x[1..n] if and only if v is in x[s.. d]"

(i) s=1, d=n=> the invariant is true

(ii) it remains true through the execution of the cycle body

(iii) when s=d the postcondition is obtained

Example 2
Binary search

Iterative variant 2

```
BinarySearch2(x[1..n],v)
 s ← 1
 d ← n
 WHILE s<d DO
   m ←(s+d) DIV 2
   IF v<=x[m]
   THEN d ← m
   ELSE s ← m+1
 ENDIF / ENDWHILE
 IF x[s]==v THEN RETURN True
            ELSE RETURN False
 ENDIF
```

Efficiency

Worst case: array x does not contain value v

Particular case : $n=2^m$

$$T(n) = \begin{cases} 1, & n = 1 \\ T\left(\frac{n}{2}\right) + 1 & n > 1 \end{cases}$$

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

T(n/2)=T(n/4)+1

...

T(2)=T(1)+1

T(1)=1

T(n)=lg (n)+1 O(log n)

Example 2
Binary search

Remarks

- Applying the rule of "smooth" functions, it follows that the BinarySeach2 algorithm (similar can be shown for the other variants) has the order of complexity $O(\log n)$ for any value of n

- Analysis of the efficiency of algorithms designed using reduction and division techniques can be facilitated by using the master theorem

Master Theorem

Consider the following recurrence relationship:

$$T(n) = \begin{cases} T_0, & n \leq n_c \\ kT\left(\frac{n}{m}\right) + T_{DC}(n) & n > n_c \end{cases}$$

If $T_{DC}(n)$(the time required for the division and combination steps) belongs to Θ(n^d) (d>=0) then

$$\text{T(n) belongs to} \begin{cases} \Theta(n^d) & k < m^d \\ \Theta(n^d \log(n)) & k = m^d \\ \Theta(n^{\log(k)/\log(m)}) & k > m^d \end{cases}$$

Remark

1. *m* represents the number of subproblems into which the original problem breaks down, and *k* is the number of subproblems that are actually solved

2. A similar result exists for classes O and Ω.

Master Theorem

Usefulness:

- It can be applied in the analysis of algorithms based on the technique of reduction or division
- Avoid explicitly resolving the recurrence relationship corresponding to execution time
- In many practical applications, the division (reduction) and combination stages are of polynomial complexity (so the master theorem can be applied)
- Unlike variants of explicit resolution of the recurrence relationship, it provides only the order of complexity, not the constants that intervene in estimating the execution time

Master Theorem

Example1: Maximum value of an array

$k=2$ (the initial PB is divided into two subproblems, and both subproblems need to be solved)

$m=2$ (the size of each subproblem is approximately n/2)

$d=0$ (the steps of dividing and combining results have constant cost)

Whereas: $k>m^d$ by applying the third case of the "master" theorem it follows that T(n) belongs to $\Theta(n^{\log(k)/\log(m)}) = \Theta(n)$

Master Theorem

Example 2: Binary search

$k=1$ (only one of the subproblems needs to be solved)

$m=2$ (problem size is n/2)

$d=0$ (division and combination steps have constant cost)

Whereas: $k=m^d$ by applying the second case of the "master" theorem one obtains that T(n) belongs to $O(n^d \log(n))= \Theta(\log n)$

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

	Merge Sort	Quick Sort
Divide	Depending on position	Depending on value
Combine	Merge	Concatenate

Merge Sort

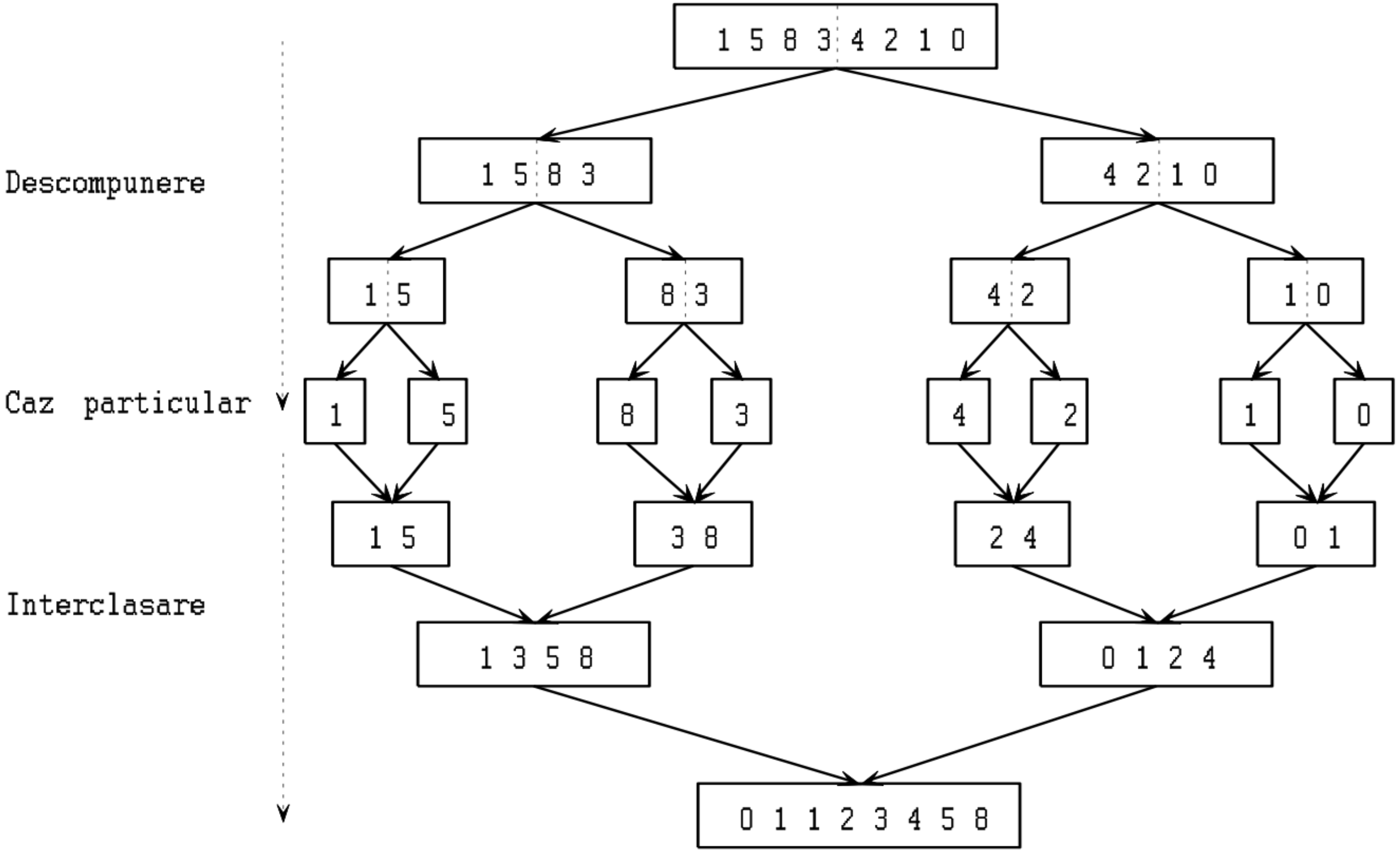
Basic idea:

- Divides x[1..n] into two subarrays x[1..[n/2]] and x[[n/2]+1..n]
- Sort each subarray
- Interclass sub-array elements x[1..[n/2]] and x[[n/2]+1..n] and build the sorted array t[1..n] . Transfer the contents of the temporary array t to x[1..n]

Remark:

- Critical value: 1 (an array containing only one item is sorted by default)
- The critical value can be greater than 1 (e.g. 10) and sorting subarrays with a number of elements less than the critical value can be done with one of the elementary alg. Sort by insert)

Merge


1 5 8 3 4 2 1 0
Descompunere
1 5 8 3
4 2 1 0
1 5
8 3
4 2
1 0
Caz particular
1
5
8
3
4
2
1
0
1 5
3 8
2 4
0 1
Interclasare
1 3 5 8
0 1 2 4
0 1 1 2 3 4 5 8

Merge Sort

Algorithm

```
MergeSort(x[s..d])
  IF s<d THEN
     m ←(s+d) DIV 2 // division
     x[s.. m] ← MergeSort(x[s.. m]) //resolution
    x[m+1..d] ← MergeSort(x[m+1..d])
    x[s.. d] ← merge(x[s.. m],x[m+1..d]) //combination
  ENDIF
  RETURN x[s..d]
```

Remark

- The algorithm is called by sort(x[1..n])

Merge Sort

```
merge (x[s..m],x[m+1..d])
 i ← s; j ← m+1; k ← 0;
// scroll through the subarray in parallel and at each step the
smallest element is transferred
  WHILE i<=m AND j<=d DO
      k ← k+1
      IF x[i]<=x[j]  THEN
        t[k] ← x[i];   i ← i+1
      ELSE
        t[k] ← x[j];   j ← j+1
       ENDIF
 ENDWHILE
// transfer any remaining elements to the first sub-array
WHILE i<=m DO
    k ← k+1
    t[k] ← x[i];   i ← i+1
ENDWHILE
// transfer any remaining elements to the second sub-array
WHILE j<=d DO
   k ← k+1
   t[k] ← x[j];   j ← j+1
ENDWHILE
RETURN t[1..k]
```

Merge Sort

- Merge is a processing that can be used to construct a sorted array from two other sorted arrays (a[1..p], b[1..q])
- A variant of merge based on sentinel values: add two values greater than array elements a[p+1]=inf , b[q+1]=inf

```
merge(a[1..p],b[1..q])
 a[p+1] ← ∞ ; b[q+1] ← ∞
 i ← 1; j ← 1;
 FOR k ← 1,p+q DO
   IF a[i]<=b[j]  THEN   c[k] ← a[i];  i ← i+1
                  ELSE   c[k] ← b[j];   j ← j+1
   ENDIF
 ENDFOR
 RETURN c[1..p+q]
```

Analysis of the efficiency for merge algorithm

Dominant operation: comparison

T(p,q)=p+q

In the sorting algorithm (p=[n/2], q=n-[n/2]):

T(n)<=[n/2]+n-[n/2]=n

So T(n) belongs to O(n) (merge algorithm has of linear complexity)

Merge Sort

Analysis of merge sort

$$T(n) = \begin{cases} 0, & n = 1 \\ T\left(\left[\frac{n}{2}\right]\right) + T\left(n - \left[\frac{n}{2}\right]\right) + T_M(n) & n > 1 \end{cases}$$

Whereas: k=2, m=2, d=1 ($T_M(n)$ belongs to O(n)) results (using the second case of the "master" theorem) that T(n) belongs to O(nlgn).

Remarks

- The main disadvantage of merge sorting is that it uses an additional array equal with the initial array size
- If the inequality <= is used in the merge step, then sorting by MergeSort is stable

Next course

- The next course will be about...
- ... Quick sort
- ... and other applications of the division
- technique

Q&A