European Funds
Knowledge Education Development
Wrocław University of Science and Technology
European Union
European Social Fund
ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej

Data Structures and Algorithms – W12

Fundamental Techniques

Contents

- Fundamental techniques for solving problems:
  - „Devide and conquer”
  - Dynamic programming
  - Greedy algorithm
  - Other techniques
- Interesting problems

Divide and conquer

- A problem input (instance) is **divided** according to some criteria **into a set of smaller** inputs to the same problem. The problem is then **solved for each of these smaller** inputs, either recursively by further division into smaller inputs or by invoking an ad hoc or a priori solution. Finally, **the solution** for the **original input** is obtained by expressing it in some form as **a combination of the solution** for these **smaller inputs**.
- *Ad hoc solution* are often invoked when the input size is smaller than some preassigned *threshold* value.
- Subproblems are independent!

Divide and conquer – common alg.

```
procedure Divide_and_conquer(I,J)
  Input:  I (an input to the given problem)
  output: J (a solution to the given problem
  corresponding to the input I)
  if I is_Known then
     assign the a priori or ad hoc solution for I to J
  else
     Divide(I, I_1,…,I_m)  // m may depend on the input I
     for i=1 to m do
        Divide_and_conquer(I_i,J_i)
     endfor
     Combine(J_1,…,J_m,J)
  endif
end Divide_and_conquer
```

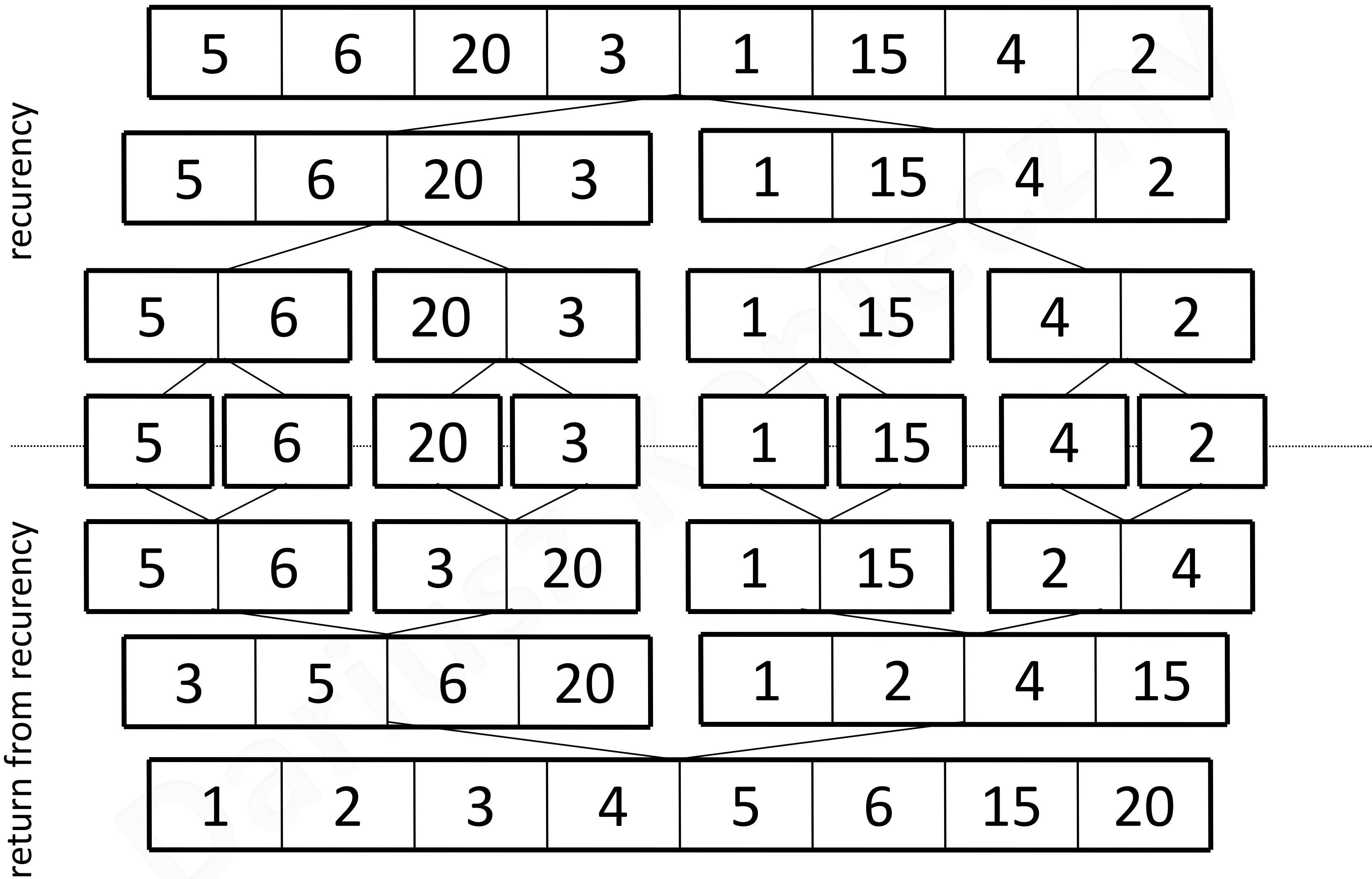
Divide and conquer – mergesort

Merge-sort idea:

1. Divide the input part of table into two equal (equally likely) parts A and B
2. Sort part A
3. Sort part B
4. Merge parts A and B, knowing that this part are sorted

– Stop the recurrence if size of an input part is equal 1. The table with only one element is always sorted.

Mergesort – an example



Dynaming programming

- **Dynamic programming** is similar to divide-and-conquer in the sense, that it is based on *recursive division of problem* instances into *smaller or simpler* problem instances.. However, whereas divide-and-conquer algorithms often use a *top-down* resolution method, DP algorithms invariably proceed by solving ***all the simplest*** problem instances ***before*** combining then into ***more complicated*** problem instances in a ***bottom-up*** fashion.
- Unlike in divide-and-conquer the *subproblems* **share** a *subsubproblems*

Dynaming programming – LCS problem

Longest common subsequence (LCS)

- Let A be a sequence $A=a_0a_1...a_{n-1}$. A subsequence of A is a sequence $T = a_{i_0} a_{i_1} \ldots a_{i_{k-1}}$ where $0 \le i_0 < i_1 < \ldots < i_{k-1} < n$
- example „samples" -> „sms","ss", „mp"
- If we have two sequence $A=a_0a_1...a_{n-1}$ and $B=b_0b_1...b_{m-1}$ we a looking for a longest common substring C, which is a subsequence of A and B.

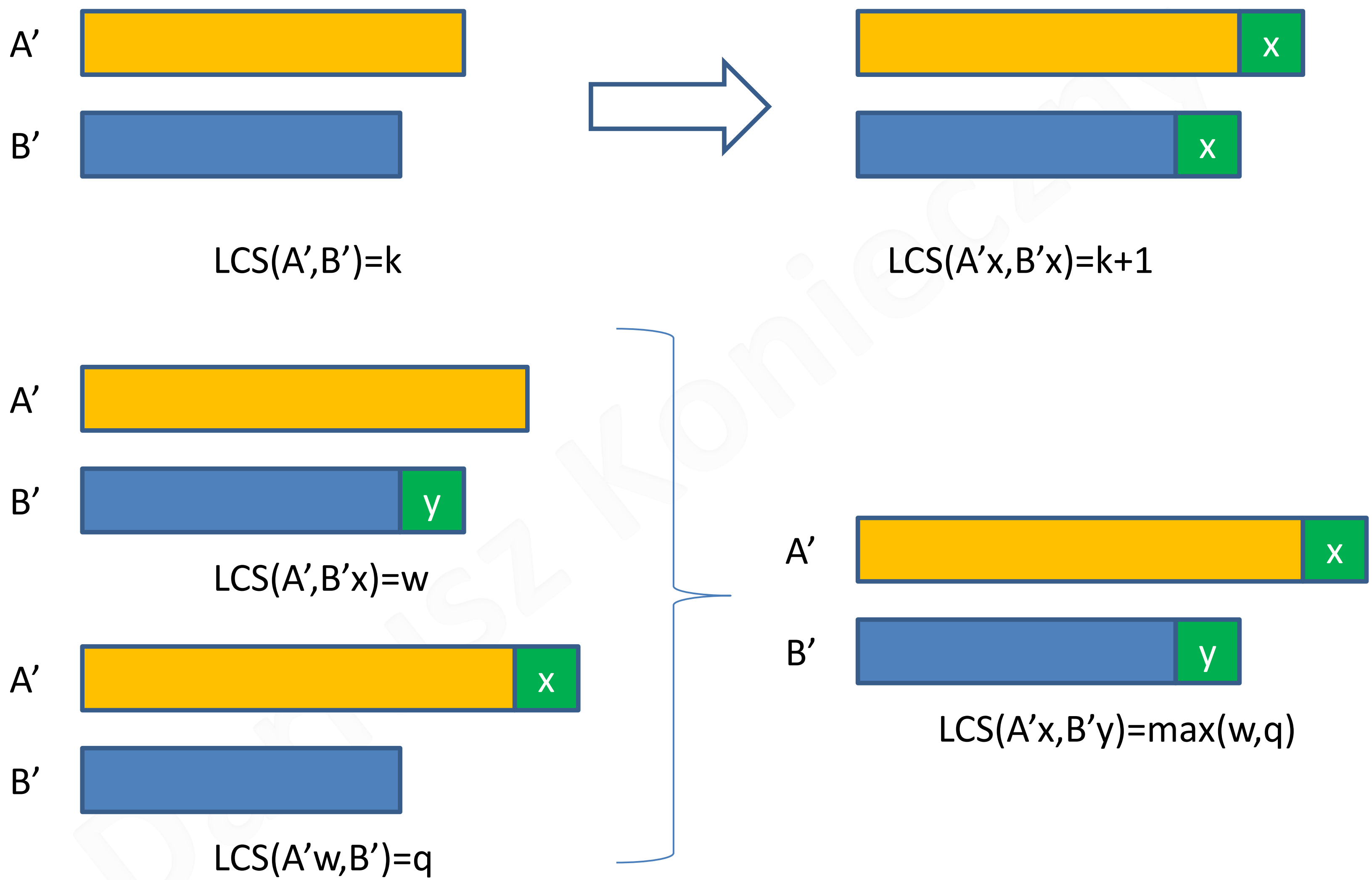
LCS – solution rules

- In real we will compute **the length of the LCS**. Let LCS[i,j] will be the length of longest common subsequence of sequences $A'=a_0a_1...a_{i-1}$ and $B'=b_0b_1...b_{j-1}$

$$LCS[i,j]=\begin{cases} 0 & if \quad i=0 \; or \; j=0 \\ LCS[i-1,j-1]+1 & if \quad a_{i-1}=b_{j-1} \\ \max(LCS[i,j-1],LCS[i-1,j]) & otherwise \end{cases}$$

- We can compute the equation recursively, but better way is to use an array and compute the values of LCS row by row from 0 to m-1 and for every row cells from 0 to n-1

LCS – solution rules



LCS – example 1/2

- A=„abbaa”
- B=„bababab”
- n=5
- m=7

			b	a	b	a	b	a	b
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
a	1	0	0	1	1	1	1	1	1
b	2	0	1	1					
b	3								
a	4								
a	5								

- LCS[n,m]=

LCS – example 2/2

- A=„abbaa”
- B=„bababab”
- n=5
- m=7

- LCS[n,m]=4

O(n*m)

$O(n^2)$

			b	a	b	a	b	a	b
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
a	1	0	0	1	1	1	1	1	1
b	2	0	1	1	2	2	2	2	2
b	3	0	1	1	2	2	3	3	3
a	4	0	1	2	2	3	3	4	4
a	5	0	1	2	2	3	3	4	4

Greedy algorithm

The **greedy method** for solving optimization problems follows the philosophy of *greedily maximizing (or minimizing*) ***short-term gain*** and hoping for the best without regard to long term consequences.

- Making decision based on optimizing short-term gain **may not lead** to solution that is **optimal**. So that you always **need to prove** that greedy solutions are **indeed optimal**
- <u>**Advance**</u>: Algorithms based on the greedy method are usually **very simple, easy to code, and efficient**
- <u>**Disadvance**</u>: when we uses the greedy method in algorithm to solve a problem, we **often end up with less-than-optimal result**.
- <u>**Advance**</u>: for some **important problem** the greedy method does **yield optimal results** (it is **proved**)!
- <u>**Advance**</u>: in some **important problems**, the greedy method yields results that are **not optimal** but in some sense are **good approximations** to optimal results.

Greedy algorithm – common alg.

```
procedure Greedy(S,Solution)
input: S (base set) // it is assumed that there is an associated objective
                   // function f defined on (possibly ordered) subsets of S
output: Solution (an ordered subset of S that potentially optimizes
                 the objective function f, or a message that Greedy
                 doesn’t even produce a solution, optimal or not)
  PartialSolution = Ø    // initialize the partial solution to be empty
  R=S
  while PartialSolution is not a solution and R!=0 do
    x=GreedySelect(R)
    R=R\{x}
    if PartialSolution U {x} is feasible then
        PartialSolution= PartialSolution ∪ {x}
    endif
  endwhile
  if PartialSolution is a solution then
      Solution=PartialSolution
  else
      write(„Greedy fails to produce a solution”)
  endif
end Greedy
```

Greedy algorithm – making change

Making change - suppose we have just purchased something and the salesperson wishes to give back exact change using the ***fewest number*** of coins. We assume that there is a sufficient amount of coins to make a change in any manner whatsoever.

- A greedy algorithm for making changes uses as many coins of the largest denomination as possible, then uses as many coins of the next largest denomination, and so forth.

Making change – an example(1)

possible coins: 1gr, 2gr, 5gr, 10gr, 20gr, 50gr

- total change=97gr
- change 50gr, rest=47gr
- change 20gr, rest=27gr
- change 20gr, rest=7gr
- change 5gr, rest=2gr
- change 2gr, rest=0gr
- number of coins = 5

Making change – an example(2)

possible coins: 1gr, 4gr, 5gr, 10gr, 40gr, 50gr

- total change=88gr
- change 50gr, rest=38gr
- change 10gr, rest=28gr
- change 10gr, rest=18gr
- change 10gr, rest=8gr
- change 5gr, rest=3gr
- change 1gr, rest=2gr
- change 1gr, rest=1gr
- change 1gr, rest=0gr
- number of coins = 8

- the best solution 88gr=40gr+40gr+4gr+4gr, number of coins = 4

By extension the **short-term** of gain we can improve the algorithm to be correct. Will it be still greedy algorithm…?

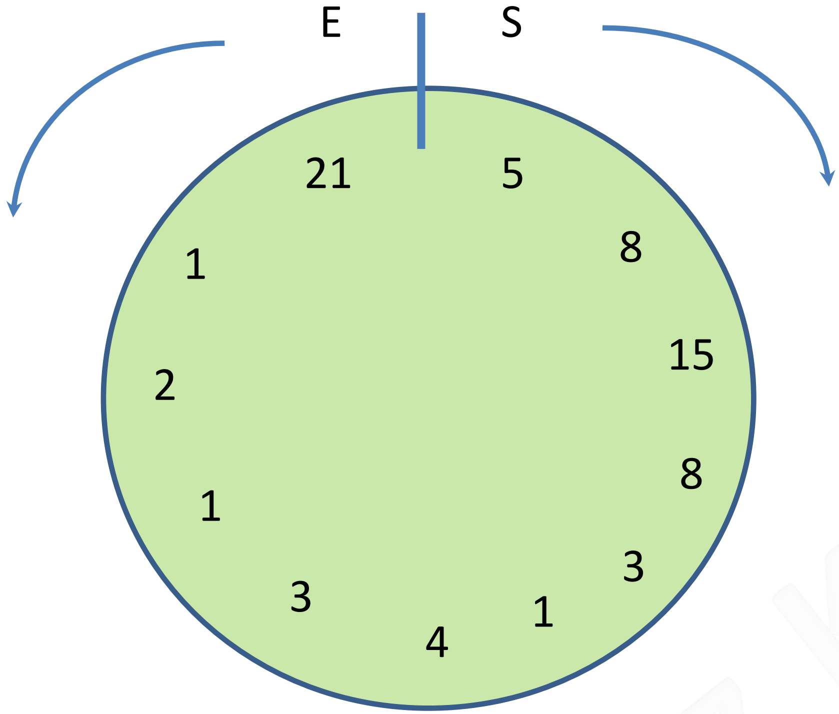
Other techniques

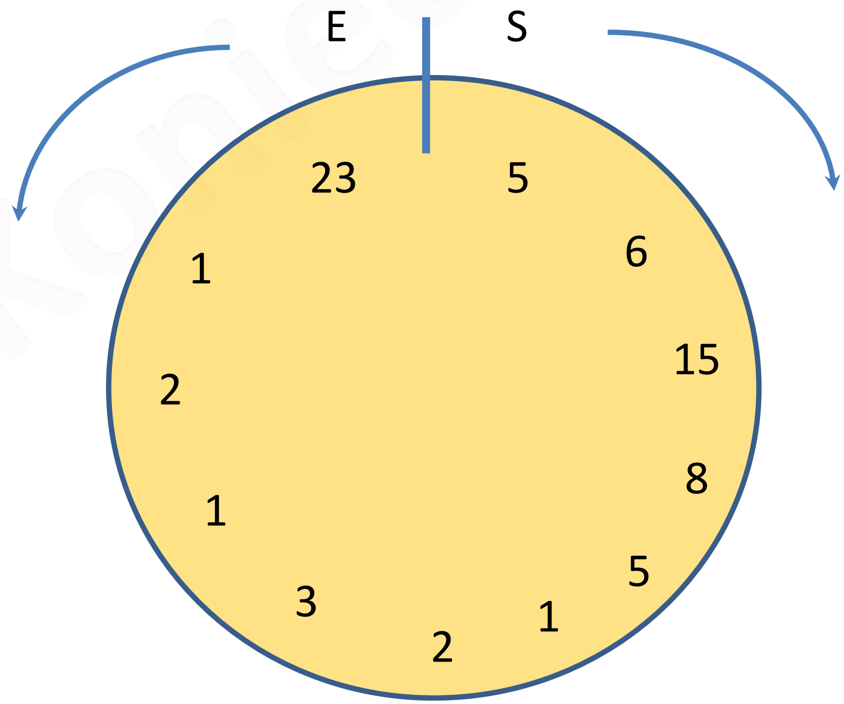
- Brute force - exhaustive search
  - Generating all solutions, rejecting non-feasible and out of the other optimal selection.
  - If you are just looking for an algorithm, useful to create the correct tests.
- Search algorithms (of space of states)
  - Similar to the exhaustive search, however, it can reject sets of solutions that will never lead to a correct solution.
- Sweep line algorithm
  - Rather, in graphical algorithms, after the initial processing of data, information is gathered for solution by adding, for example, in order: from left to right, from bottom to top
- Etc.

Example problems

- http://livearchive.onlinejudge.org/
- https://icpcarchive.ecs.baylor.edu/
- ->Browse Problems->ICPC Archive Volumes
  - **2535 - Magnificent Meatballs**
  - **2122 - Recognizing S Expressions**
  - **2487 - Lollies**
  - **3390 - Pascal's Travels**

Magnificent Meatballs - examples





S-expression examples

- Which are S-expressions?
  - t
  - #
  - tt
  - t,t
  - (t,t)
  - ((a,b))
  - (t,a,b)
  - ((A,a),b)
  - ((a,b),c
  - ((a,b),(c,g))
  - (((t,u),w),h)
  - (q,(w,(e,r)))
  - [x,y]

Lollies - examples

day	lollies	delay	solution
1	3	4	
2	5	8	
3	4	6	
4	2	3	
5	3	7	
6	4	3	
7	4	3	
8	3	1	
9	2	3	