Algorithms and Computability

Lecture 1
Intro & Dynamic Programming

SW6/DVML8 spring 2025
Simonas Šaltenis

AALBORG UNIVERSITY
DENMARK

People

- Lecturers:
  - First 6 lectures: Simonas Šaltenis
    - Email: simas@cs.aau.dk
    - Office: 3.2.05
  - Last 7 lectures: Christian Schilling
    - Email: christianms@cs.aau.dk
    - Office: 1.2.14

- Teaching assistant:
  - Victor Doré Hansen vdha20@student.aau.dk

Location, Time, Structure

- Location: 0.1.95 + Fib16, FRB7H, Kst3,...
- Time: Tuesdays, 12:30–14:15 (Exercises: 14:30–16:15)
  - Self-studies and mini-projects other days – see the schedule.
  - Please, check the schedule on Moodle for any changes.

- A total of 16 sessions:
  - 13 regular sessions + 3 self-study sessions
  - A regular session = 2-hour lecture + 2-hour exercises
  - A self-study exercise session = 4 hours of exercises

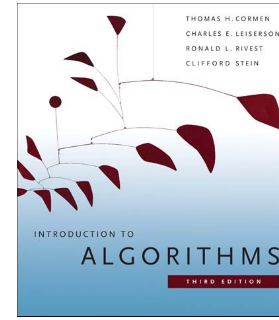
Workload

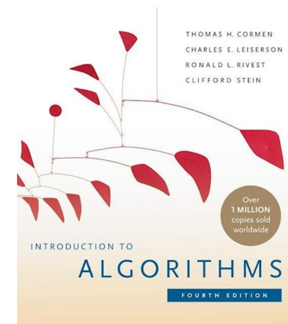
- This is a 5 ECTS course =~ 150 hours of your effort
  - 13 regular sessions:
    - 2h lecture + 2h exercises + 3.5h preparing. In total, 13*(2+2+3.5)=97.5h
  - 3 self-studies:
    - 4h solving exercise + 3h preparing/feedback. In total = 21h

- Exam and preparation for it =~ 31.5h
- In total: 97.5+21+31.5 = 150h

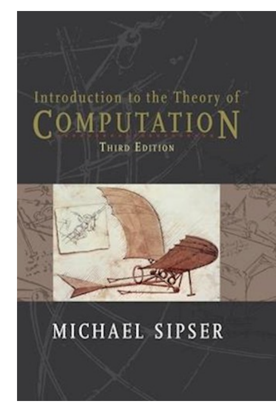
Textbook

- First six lectures:
  - T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, **3rd edition**, The MIT Press. ISBN:9780262533058
  - ...or **4th edition.** ISBN: 9780262046305
  - On the Moodle, I use CLRS for both editions, or CLRS3/CLRS4 when there is a difference.
  - Additional notes and videos.





- Last seven lectures:
  - Michael Sipser. *Introduction to the Theory of Computation*, **Third International Edition**. Thomson Course Technology. ISBN: 9781133187790



Advice, Exam

- **Prepare** for lectures: read, watch videos.
- Be active during lectures, have paper and pen – there will be mini-exercises/quizzes.
- **Exercises, self-studies, and mini-projects** are very important:
  - The exam will consist of a set of exercises / questions
  - Some parts of exam exercises can directly relate to selected parts of self-studies/mini-projects.
  - Make sure you understand all exercises by YOURSELF even if working on them in a group.
- Your feedback, positive and negative, is always welcome!

- The **exam** will be a 4-hour Moodle-based digital exam with notes and books.

What is it about?

- The course is about ***algorithms and algorithmic problems***
  - The first six lectures cover some selected *algorithms*, algorithm *design techniques*, and algorithm *analysis techniques* for problems with known efficient algorithms
    - We continue where the AD course left off, but focus a bit more on the design of algorithms rather than just understanding them “as is”.
  - We focus on the *efficiency of algorithms*, i.e., on the *upper bounds*.
  - The last seven lectures focuse on characterizing how hard the *problems* are:
    - How to formalize computation?
    - Do all problems have an algorithmic solution?
    - How do we show that one problem is as hard as the other?
    - We focus on *hardness of problems*, i.e., on the *lower bounds.*

Course content

- Lecture 1: Dynamic programming
- Lecture 2: Greedy algorithms
- Lecture 3: Maximum flow
- Lecture 4: External-memory algorithms
- Lecture 5: Parallel algorithms
- Lecture 6: Amortized analysis

Self-study

Self-study

- Lecture 7: Turing machines
- Lecture 8: The Church-Turing thesis
- Lecture 9: Decidability
- Lecture 10: Reducibility
- Lecture 11: Time complexity – the complexity class P
- Lecture 12: NP and NP-completeness
- Lecture 13: NP-complete problems

Self-study

Mini-quiz 1

1.2. *(3 points)* $700 \cdot n^2 + 999 \cdot n^2 \lg n + 0.1 \cdot n^2 \lg^2 n$ is:

☐ **a)** $\Theta(n^2 \lg n)$ ☐ **b)** $\Omega(n^2 \lg n)$ ☐ **c)** $\Theta(n^2)$ ☐ **d)** $\Theta(n^2 \cdot \lg^2 n)$

- Go to Socrative and vote (the link is also on course Moodle)
  - Multiple choices could be correct

Mini-quiz 2

- From the reading material for today: "*… would take $\omega(1)$ time…*" What does it mean?
  - A: Would take *constant time* (not dependent on problem size)
  - B: Would take *more than constant time*
  - C: Would take *constant time or more*

- Go to Socrative and vote (the link is also on course Moodle)

Mini-quiz 3

- Have you prepared for the lecture today? (be honest)
  - A: Read everything and watched the two videos
  - B: Read some of it and watched the two videos
  - C: Did not read anything, but watched the two videos
  - D: Watched some (parts) of the videos and read some of it
  - E: Did not have time/energy/desire to prepare at all

- Go to Socrative and vote (the link is also on course Moodle)

Dynamic Programming

- Goals of this lecture:
  - *to understand the **principles** of dynamic programming;*
  - *to understand how an algorithm for **edit distance** works;*
  - *to be able to **apply** the dynamic programming algorithm design technique.*

Optimization problems

- Many problems can be framed as *optimization problems*:
  - Find the *shortest route* from *A* to *B.*
  - Find the *items* that give *most value* and can fit into a knapsack.
- Two things that we need to find:
  - *Compute* the optimum *value:*
    - Length of a route
    - Total value of items in the knapsack.
  - *Construct an object* that has that optimum value (i.e., proof of the value):
    - Route
    - The set of items

Dynamic programming

- *Dynamic programming*:
  - A powerful technique to solve *optimization problems*

- Structure:
  - To arrive at an optimal solution *a number of choices* are made
  - Each *choice generates* a number of *sub-problems*
  - Which choice to make is decided by looking at all possible choices and the solutions to sub-problems that each choice generates.
  - The solution to a specific sub-problem is used many times in the algorithm
    - Subproblems are *overlapping*
  - *First*, think how to compute the value of a variable that we optimize,
    - *Then*, augment your algorithm to remember the choices made.
    - *Finally*, the choices can be traced back to build an optimal solution corresponding to an optimal value.

DP algorithm design roadmap

- Construction:
  - *Which choices have to be considered in each step of the algorithm?*
  - *What are the sub-problems? Which parameters define each sub-problem?*
  - *How are the trivial sub-problems solved?*
  - (*In which order do we have to solve the sub-problems?*)
    - Or write a *memoized* version of the algorithm

Recurrence for the optimal value

  - *Remember the (optimal) choices made*
  - *Use the remembered choices to construct a solution*

Constructing a solution

- Analysis:
  - *How many different sub-problems are there in total?*
  - *How many choices have to be considered in each step of the algorithm?*

Edit Distance

- Problem definition:
  - Two strings: $s[1..m]$, and $t[1..n]$
  - Find *edit distance* $dist(s,t)$– the smallest number of edit operations that turns s into t
  - Edit operations:
    - **Replace** one letter with another
    - **Delete** one letter
    - **Insert** one letter

  - Example:

ghost	delete **g**
host	insert **u**
houst	replace **t** by **e**
house	

Sub-problems

- What are the sub-problems?
  - *Goal* 1: To have as few sub-problems as possible
  - *Goal* 2: Solution to the sub-problem should be possible by combining solutions to smaller sub-problems.

- Sub-problem:
  - $d_{i,j} = dist\ (s\ [1..i\ ],\ t\ [1..j\ ])$
  - Then $dist\ (s,\ t\ ) = d_{m,n}$

Making a choice

- *How can we solve a sub-problem by looking at solutions of smaller sub-problems to make a choice?*
  - Let's look at the last symbol: $s[i]$ and $t[j]$. There are three options, do whatever is cheaper:
    - If $s[i] = t[j]$, then turn $s[1..i-1]$ to $t[1..j-1]$, else **replace** $s[i]$ by $t[j]$ and turn $s[1..i-1]$ to $t[1..j-1]$
    - **Delete** $s[i]$ and turn $s[1..i-1]$ to $t[1..j]$
    - **Insert** insert $t[j]$ at the end of $s[1..i]$ and turn $s[1..i]$ to $t[1..j-1]$

Recurrence

$$d_{i,j} = \min \begin{cases} d_{i-1,j-1} + \begin{cases} 0 & \text{if } s[i] = t[j] \\ 1 & \text{else} \end{cases} \\ d_{i-1,j} + 1 \\ d_{i,j-1} + 1 \end{cases}$$

- *How do we solve trivial sub-problems?*
  - To turn empty string to *t* [1..*j*], do *j* **insert**s
  - To turn *s* [1..*i*] to empty string, do *i* **delete**s
- *(In which order do we have to solve the sub-problems?)*

Algorithm, memoized

```
EditDistance(s[1..m], t[1..n])
for i =  0 to m do
   for j =  0 to n do
      dist[i, j] = ∞
return EditDistR(s, t, m, n)

EditDistR(s, t, i, j)
if dist[i,j] == ∞ then
   if j == 0 then dist[i,j] = i
   else if i == 0 then dist[i,j] = j
   else
      if s[i] == t[j] then
         dist[i,j] = min(EditDistR(s,t,i-1,j-1),
                         EditDistR(s,t,i-1,j)+1,
                         EditDistR(s,t,i,j-1)+1)
      else
       dist[i,j] = 1 + min(EditDistR(s,t,i-1,j-1),
                           EditDistR(s,t,i-1,j),
                           EditDistR(s,t,i,j-1))
return dist[i,j]
```

$$d_{i,j} = \min \begin{cases} d_{i-1,j-1} + \begin{cases} 0 & \text{if } s[i] = t[j] \\ 1 & \text{else} \end{cases} \\ d_{i-1,j} + 1 \\ d_{i,j-1} + 1 \end{cases}$$

Algorithm

```
EditDistance(s[1..m], t[1..n])
for i = 0 to m do dist[i,0] = i
for j = 0 to n do dist[0,j] = j
for i = 1 to m do
   for j = 1 to n do
      if s[i] = t[j] then
         dist[i,j] = min(dist[i-1,j-1], dist[i-1,j]+1,
                         dist[i,j-1]+1)
      else
         dist[i,j] = 1 + min(dist[i-1,j-1], dist[i-1,j],
                         dist[i,j-1])
return dist[m,n]
```

- *What is the running time of this algorithm?*
- *How do we modify it to remember the edit operations?*

Let’s run the algorithm

$$d_{i,j}=\min\begin{cases} d_{i-1,j-1}+\begin{cases}0 & \text{if } s[i]=t[j]\\ 1 & \text{else}\end{cases}\\ d_{i-1,j}+1\\ d_{i,j-1}+1\end{cases}$$

			G	H	O	S	T
	j\i	0	1	2	3	4	5
	0	0	1_D	2_D	3_D	4_D	5_D
H	1	1_I	1_R				
O	2	2_I					
U	3	3_I					
S	4	4_I					
E	5	5_I					

I : insert
D: delete
R: replace
C: do nothing

Let’s run the algorithm

$$d_{i,j}=\min\begin{cases} d_{i-1,j-1}+\begin{cases}0 & \text{if } s[i]=t[j]\\ 1 & \text{else}\end{cases}\\ d_{i-1,j}+1\\ d_{i,j-1}+1\end{cases}$$

			G	H	O	S	T
	j\i	0	1	2	3	4	5
	0	0	1_D	2_D	3_D	4_D	5_D
H	1	1_I	1_R	1_C			
O	2	2_I					
U	3	3_I					

I : insert
D: delete
R: replace
C: do nothing

- Fill the next cell!
- Go to Socrative and write in your answer (the link is also on course Moodle)

Let’s run the algorithm

$$d_{i,j}=\min\begin{cases} d_{i-1,j-1}+\begin{cases}0 & \text{if } s[i]=t[j]\\ 1 & \text{else}\end{cases}\\ d_{i-1,j}+1\\ d_{i,j-1}+1\end{cases}$$

			G	H	O	S	T
	j\i	0	1	2	3	4	5
	0	0	1_D	2_D	3_D	4_D	5_D
H	1	1_I	1_R	1_C	2_D		
O	2	2_I					
U	3	3_I					
S	4	4_I					
E	5	5_I					

I : insert
D: delete
R: replace
C: do nothing

Elements of Dynamic Programming

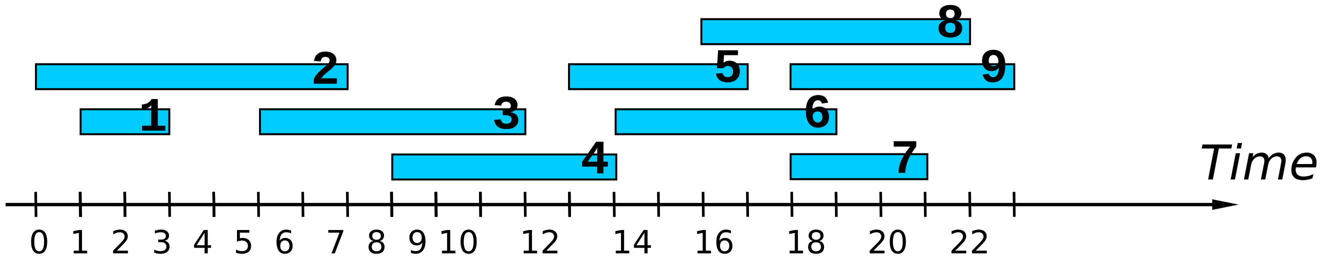
- Dynamic programming is used for optimization problems
  - A number of choices have to be made to arrive at an optimal solution
  - At each step, consider all possible choices and solutions to sub-problems induced by these choices (compare to greedy algorithms)
  - The order of solving of the sub-problems is important – from smaller to larger
- Usually a table of sub-problem solutions is used

Elements of Dynamic Programming

- To be sure that the algorithm finds an optimal solution, the *optimal sub-structure* property has to hold
  - the simple “cut-and-paste” argument usually works:
    - **If** an optimal solution includes a choice that we consider **then** it includes optimal solutions to the subproblems that this choice generates.
  - but not always! Longest simple unweighted path example – no optimal sub-structure!
    - The subproblems have to be *independent.*

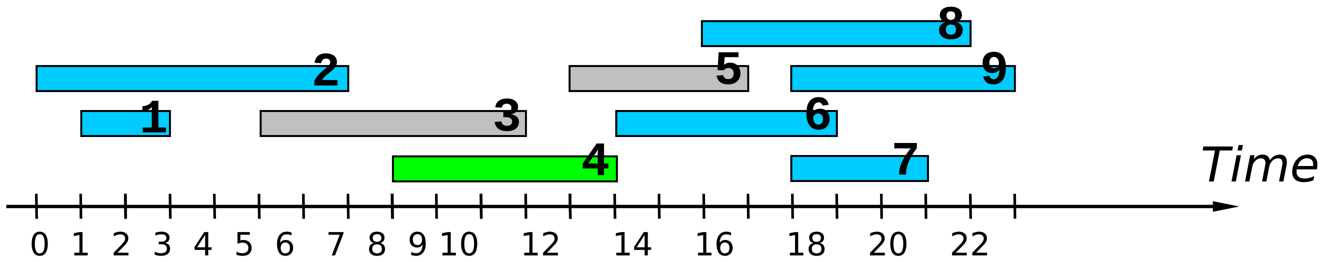
Activity-Selection Problem

- Input:
  - A set of n activities, each with start and end times: $A[i].s$ and $A[i].f$. The activity lasts during the period $[A[i].s, A[i].f)$
- Output:
  - The ***largest*** subset of mutually *compatible* activities
    - Activities are compatible if their intervals do not intersect



“Straight-forward” solution

- Let’s just pick (schedule) one activity $A[k]$
  - This generates two set’s of activities compatible with it: *Before*(k), *After*(k)
    - E.g., *Before*(4) = {1, 2}; *After*(4) = {6,7,8,9}



  - Solution:

$$MaxN(A) = \begin{cases} 0 & \text{if } A = \emptyset, \\ \max\limits_{a \in A} \{ MaxN(Before(a)) + MaxN(After(a)) + 1 \} & \text{if } A \neq \emptyset. \end{cases}$$

Dynamic Programming Alg.

- The recurrence results in a dynamic programming algorithm
  - Sort activities on the end time (for simplicity assume also “sentinel” activities $A[0]$ and $A[n+1]$)
  - Let S_{ij} – a set of activities after $A[i]$ and before $A[j]$ and compatible with $A[i]$ and $A[j]$.
  - Let’s have a two-dimensional array, s.t., $c[i, j] = MaxN(S_{ij})$:

$$c[i,j]=\begin{cases} 0 & \text{if } S_{ij}=\emptyset, \\ \max\limits_{a_k \in S_{ij}}\{c[i,k]+c[k,j]+1\} & \text{if } S_{ij}\neq\emptyset. \end{cases}$$

  - $MaxN(A) = MaxN(S_{0,n+1}) = c[0, n+1]$

Dynamic Programming Alg. II

- Does it really work correctly?
  - We have to prove the optimal sub-structure:
    - *If an optimal solution A to S_{ij} includes A[k], then it also includes optimal solutions to S_{ik} and S_{kj}*
    - To prove use "cut-and-paste" argument

- What is the running time of this algorithm?

Activity Selection DP Alg. 2.0

- Alternative way of thinking about it – ***binary choice***:
  - Sort activities on the start time (have "sentinel" activity $A[n+1]$ after all the other activities)
  - Let $next(i) = \min \{k \mid k > i \ \wedge \ \neg overlaps(A[i], A[k])\}$
  - The subproblem is then to schedule all the activities starting with i and after.

$$c[i] = \begin{cases} 0 & \text{if } i > n, \\ \max\left(1 + c[next(i)],\ c[i+1]\right) & \text{otherwise}. \end{cases}$$

  - $MaxN(A) = c[1]$
  - *What is the running time and space used?*