Advanced Algorithms

Lecture 1
Introduction
&
Dynamic Programming

Bin Yang
byang@cs.aau.dk

Agenda

- Introduction
- Dynamic Programming

People

- Lecturers: Bin Yang, Tung Kieu, Chenjuan Guo
  - Email: byang@cs.aau.dk
  - Office: 3.2.46
  - Homepage: http://people.cs.aau.dk/~byang/
  - **D**PW group: **Database**, Programming and Web Technologies
    - Big data, data science, data analytics (machine learning, artificial intelligence)
  - Daisy: Center for Data-Intensive Systems. http://www.daisy.aau.dk/

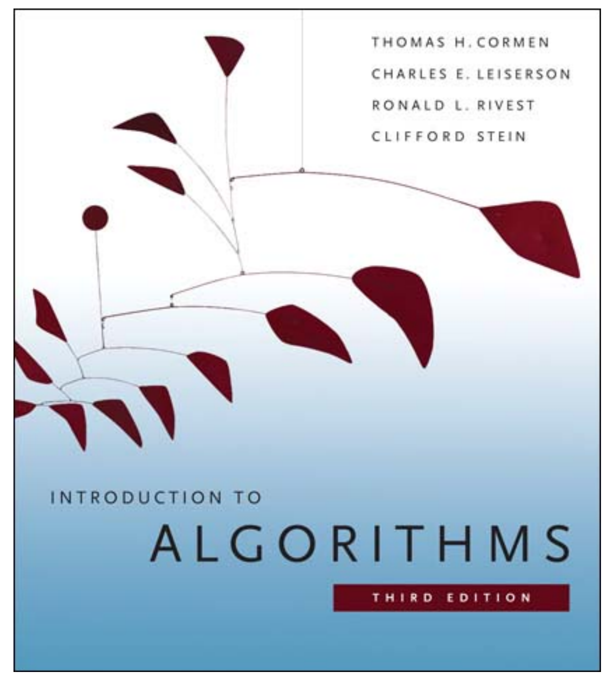
Location and Time

- Location
  - 0.1.95.
- Time
  - Tuesdays and Fridays.
  - Lectures are from 8.15 to 10.00.
  - Exercises are from 10.15 to 12.00.

- Check the full schedule on Moodle in case there are last-minute changes.

Moodle Page and Textbook

- Course page at Moodle
  - https://www.moodle.aau.dk/course/view.php?id=33044
  - Please check it frequently for notifications and updates!
- Textbook
  - "Introduction to Algorithms", 3.ed, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, The MIT Press.
  - ISBN:9780262533058
  - Abbreviation: CLRS
  - Available at the Factum book store http://ftu.dk/



Course Structure

- A total of 15 sessions
  - 12 regular sessions + 3 self-study exercise sessions
- A regular session = a lecture class + an exercise class.
- A lecture class
  - 2 * 45 minutes
  - Each lecture we focus on a specific topic.
  - During a lecture, there may be mini-quizzes.
    - Have a pen and some papers.
- An exercise class
  - 2 * 45 minutes
  - Solve exercises assigned in the session.
    - You are encouraged to work in groups.
    - TAs come by from group to group during each exercise class.
  - Get a feeling of exam questions from enough exercises!

Course Structure (2)

- A self-study exercise session = 4 hours of exercises
  - You need to do it in groups.
  - Each group ***can*** submit to TA or me one written solution no later than a week of the session.
  - TA/I will give written feedback for each of the submitted solutions.

  - I recommend that each of you solves the problems **individually** first, and then you discuss, summarize, and hand in solutions **per group**.
  - In case you cannot agree with each other, you can hand in multiple solutions for one problem.

Working hours

- This is a 5 ECTS course, where each ECTS=27.5 hours
  - 137.5 hours
- 12 regular sessions
  - 2h lecture + 2h exercises.
  - 3h reading
  - In total, 12*(2+2+3)=84h
- 3 self-study exercises sessions
  - 4h on solving the exercises.
  - 4h on checking the solutions/feedback in order to make sure you can solve each of the exercises.
  - In total, 3*(4+4)=24h
- 29.5h for preparing the exam.
- In total, 84+24+29.5=137.5h

Exam

- Individual and written, but open-book
  - Electronic devices with communication capabilities, such as laptops and mobile phones, are NOT allowed.
  - You can bring old-fashion calculators.
  - You can freely use your copies of slides from the lectures, textbooks, and other course material.
- Exam in two parts
  - A set of quizzes, to test knowledge.
    - Choose the options that you think are correct.
  - A few (1 ~ 3) open problems, each with some sub-problems, to test competences and skills.
    - Given a real world problem, write pseudo code, give complexity analysis, etc.
  - You will get a better feeling when you participate self-study exercise sessions.
- The exam will have 100 points.

Prerequisites

- AD on DAT3/SW3 or AD2 on IT7
- Let's quickly recap the intended learning outcomes of AD/AD2
- Basic mathematical concepts such as recursion, induction, concrete and abstract complexity;
  - Solving recurrences, asymptotic notation.
- Basic data structures;
  - Queues, stacks, heaps, linked lists, priority queues.
- Algorithmic principles such as searching, search trees, sorting, dynamic programming, divide-and-conquer;
  - Binary search tree, merge sort, quick sort.
- Graphs and graph algorithms such as shortest path, connected components, spanning trees.
  - BFS, DFS, MST, topological sorting, shortest path.

Mini quiz

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)** $\Theta(n^2 \cdot \lg^2 n)$

Intended Learning Outcomes (ILO)

- After taking AALG, you should acquire the following knowledge
  - Algorithm **design** techniques such as divide-and-conquer, greedy algorithms, dynamic programming, back-tracking, branch-and-bound algorithms, and plane-sweep algorithms;
  - Algorithm **analysis** techniques such as recursion, amortized analysis;
  - A collection of **core** algorithms and data structures to solve a number problems from various computer science areas: algorithms for external memory, multiple-threaded algorithms, advanced graph algorithms, heuristic search and geometric calculations;
  - There will also enter into one or more **optional subjects** in advanced algorithms, including, but not limited to: *approximate algorithms*, randomized algorithms, search for text, linear programming and number theoretic algorithms such as crypto-systems.

Course Content (1)

- Lecture 1: Dynamic programming (CLRS 15)
  - Principles of DP.
  - Examples: Edit distance, activity selection.
- Lecture 2: All-pairs shortest paths (CLRS 25)
  - Distance matrix and predecessor matrix.
  - Floyd-Warshall algorithm, which uses DP.
- Lecture 3: Network flow algorithms (CLRS 26)
  - Formalize flow networks, flows, maximum-flow.
  - Ford-Fulkerson algorithms.
- Lecture 4: Greedy algorithm (CLRS 16)
  - Ideas/principles of greedy algorithm.
  - Examples: Activity selection, Huffman coding.

Course content (2)

- Lecture 5: Amortized analysis (CLRS 17)
  - Understand amortized analysis, difference from average-case analysis.
  - Aggregated analysis, accounting method, potential method.
- Lectures 6, 7, 8: Computational geometry (CLRS 33 + additional references)
  - Basic geometric operations in 2D, e.g., two line segments intersecting, orientation of two line segments?
  - Sweeping algorithms.
  - Graham’s scan and Jarvis's march algorithm for convex hull.
  - Divide-and-conquer algorithm to find the closet pair of points in a set of points.
  - Range searching in d-dimensional space: kd-tree and range tree.

Course Content (3)

- Lecture 9: External-memory algorithms and data structures (CLRS 18 + additional references)
  - Balanced search trees, e.g., *B-trees* and *R-trees.*
  - External memory sorting: multi-way merge-sort algorithm.
- Lecture 10: Multi-threaded algorithms (CLRS 27)
  - Concurrency keywords: parallel, spawn, sync.
  - MT Fibonacci number computation, MT merge sort.
- Lecture 11 & 12: Algorithms for NP-complete problems (CLRS 35)
  - Approximation algorithms
  - Backtracking and branch-and-bound
  - Examples using the algorithms.
    - Vertex cover, traveling sales man.

Tips

- Your feedback is always welcome.
  - Esp. when you get confused.
  - Send me an email or drop by my office.
- Participate in every lecture.
- Play actively in every exercise class and self-study exercise session.
  - Exercises prepare you for the final exam!
  - Make sure you understand all exercises by YOURSELF after your group work in each exercise class.
- Check the course page frequently.

Agenda

- Introduction
- Dynamic Programming (DP)
  - To understand the principles of dynamic programming.
  - To understand the DP algorithm for edit distance.
  - To be able to apply the DP algorithm design technique.

Recall algorithm design techniques from AD1

- Algorithm design techniques so far:
  - Brute-force algorithms
    - Linear search
  - Incremental algorithms
    - Insertion sort
  - Algorithms that use ADTs (implemented using efficient data structures)
    - Heap sort
  - Divide-and-conquer algorithms
    - Merge sort, quick sort.
  - Dynamic-programming
    - Rod cutting
    - Top-down with memoiziation.
    - Bottom-up method.

Divide and Conquer

- If the problem size is small enough to solve it in a straightforward manner, solve it.
- Otherwise, meaning that the input size is too large to deal with in a straightforward manner, do the following
  - **Divide**: Divide the problem into two or more ***disjoint*** sub-problems.
  - **Conquer**: Use divide-and-conquer recursively to solve the sub-problems.
  - **Combine**: Take the solutions to the sub-problems and combine these solutions into a solution for the original problem.

Merge Sort

```
Merge-Sort(A, p, r)
    if p < r then
        q←(p+r)/2
        Merge-Sort(A, p, q)
        Merge-Sort(A, q+1, r)
        Merge(A, p, q, r)
```

- Mini-quiz: do you still recall the **recurrence** of merge sort?
  - A **recurrence** is an equation or inequality that describes a function in terms of its value on smaller inputs.
- T(n)=
  - $\Theta(1)$ if n=1
  - $2T(n/2) + \Theta(n)$ if n>1

Dynamic programming

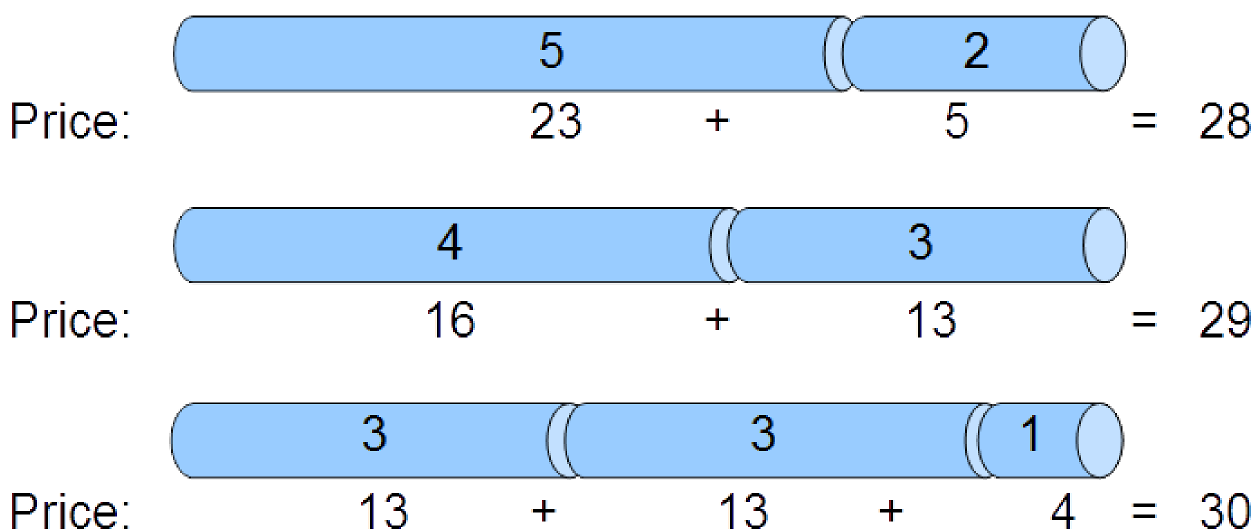
- A powerful technique to solve ***optimization problems***.
- An optimization problem can have many possible solutions, each solution has a value, and we wish to find a solution with the optimal (i.e., minimum or maximum) value.
- An algorithm should compute *the* **optimal value** plus, if needed, *an* **optimal solution**.

Rod cutting

- A steel rod of length *n* should be cut and sold in pieces.
- Pieces sold only in integer sizes according to a price table *P*[1..*n*].
- *Goal*: cut up the rod to maximize profit.

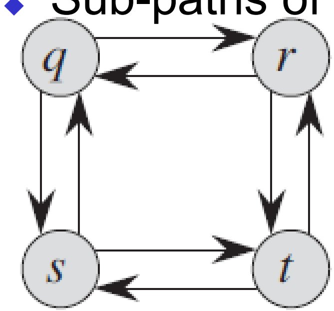
Length	1	2	3	4	5	6	7
Price	4	5	13	16	23	24	27

Max profit (optimal value): 31
Optimal cut (optimal solution): 1, 1, 5



Two key characteristics of DP

- Overlapping sub-problems
  - Sub-problems share sub-sub-problems.
  - A divide-and-conquer algorithm does more work than necessary, as it needs to repeatedly solve the common sub-sub-problems.
- Optimal substructure
  - The optimal solution to a problem incorporates optimal solutions to sub-problems.
  - Un-weighted shortest path (YES)
    - Shortest path ***A* = <*q, r, t*>** from q to t.
    - Sub-paths of ***A*,** <q, r> and <r, t>, are also the shortest paths.
  - Un-weighted longest simple path. (NO)
    - Longest path ***B* = <*q, r, t*>** from q to t.
    - Sub-paths of ***B*,** <q, r> and <r, t>, may not be the longest paths.



<q, s, t, r>

<r, q, s, t>

Two approaches of DP

- Top-down with memoization
  - Solve each sub-problem only ***once*** and **store** the answers to the solved sub-problems in a table.
  - Next time, when you need to solve a solved sub-problem, just look up the table to get the answer.
- Bottom-up without recursion.
  - Depending on some natural notion on the **size** of a sub-problem.
  - Solving any particular sub-problem depends only on solving **smaller** sub-problems.
  - Sort the sub-problems by size and solve them in size order, smallest first. And save the solutions.
- Pros and cons
  - Both should have the same asymptotic running time.
  - If all sub-problems must be solved, memoization (recursion) is usually slower (by a constant factor) than Bottom-up (loops).
  - If not all sub-problems need to be solved, memoization only solves the necessary ones.

Structure of DP

- Construction:
  - *What are the sub-problems?*
  - *Which choices have to be considered in order to solve a sub-problem?*
  - *How are the trivial sub-problems solved?*
  - Write a *memoized* version of the algorithm o*r in which order do we have to solve the sub-problems (bottom-up)*
  - *Remember the (optimal) choices made*
  - *Use the remembered choices to construct a solution*

Recurrence for the optimal value

Constructing a solution

- Analysis:
  - *How many different sub-problems are there in total?*
  - *How many choices have to be considered when solving each sub-problem?*

Agenda

- Introduction
- Dynamic Programming
  - To understand the principles of dynamic programming.
  - To understand the DP algorithm for edit distance.
  - To be able to apply the DP algorithm design technique.

Edit distance

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

- Example: let’s turn “ghost” to “house”
  - **ghost** delete **g**
  - **host** insert **u**
  - **houst** replace **t** by **e**
  - **house**

Two cases

- The last letters in s and t are different, e.g., s=milk t=windy
  - Option 1: Replace k by y, dist(s, t)=$\boxed{\text{dist(mil, wind)}}$+1
  - Option 2: Delete k, dist(s, t) = $\boxed{\text{dist(mil, windy)}}$+1
  - Option 3: Insert y in the end of s, dist(s, t) = dist(milk, wind)+1
  - dist(s, t) = min (dist(mil, wind)+1, dist(mil, windy)+1, dist(milk, wind)+1)
- The last letters in s and t are the same, e.g., s=milk t=link
  - Option 1: Keep k, dist(s, t)=dist(mil, lin)
  - Option 2: Delete k, dist(s, t) = dist(mil, link)+1
  - Option 3: Insert k in the end, dist(s, t) = dist(milk, lin)+1
  - dist(s, t) = min (dist(mil, lin), dist(mil, link)+1, dist(milk, lin)+1)
- Optimal sub-structure for edit distance?
  - YES! The optimal solution to a problem incorporates optimal solutions to sub-problems.
  - Formal proof: see Lecture Material on Moodle.

Sub-problems

- Sub-problem:
  - $d_{i,j} = dist\ (s\ [1..i\ ],\ t\ [1..j\ ])$
- Then $dist\ (s,\ t\ ) = d_{m,n}$
- Let's look at the last symbol: *s* [*i*] and *t* [*j*]. There are three options, do whatever is the cheapest:
- **Option 1:**
  - **If** *s* [*i*] = *t* [*j*], then turn *s* [1..*i*-1] to *t* [1..*j*-1]
    - milk, link: $d_{i,j}= d_{i-1,j-1}$
  - **Else replace** s[*i*] by *t* [*j*] and turn *s* [1..*i*-1] to *t* [1..*j*-1]
    - milk, windy; mily, windy: $d_{i,j}= 1+ d_{i-1,j-1}$
- **Option 2: Delete** *s* [*i*] and turn *s* [1..*i*-1] to *t* [1..*j*]
  - milk, windy; mil, windy: $d_{i,j}= 1+ d_{i-1,j}$
- **Option 3: Insert** *t* [*j*] at the end of *s* [1..*i*] and turn *s* [1..*i*] to *t* [1..*j*-1]
  - milk, windy; milky, windy: $d_{i,j}= 1+ d_{i,j-1}$

Recurrence, optimal substructure

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

Do nothing

replace s[*i*] by *t* [*j*]

delete *s* [*i*]

insert *t* [*j*] at the end of s[i]

- *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
- Do the sub-problems overlap each other?

DP Algorithm, memoization

```
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)
```

Initialization

```
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]
```

Trivial sub-problems: i deletes and j inserts

delete *s* [*i*]
insert *t* [*j*]

Replace *s* [*i*] by t[j]
delete *s* [*i*
insert *t* [*j*]

Time Complexity

- Analysis
  - *How many different sub-problems are there in total?*
    - *n*m*
  - *How many choices have to be considered when solving each sub-problem?*
    - *3 (copy/replace, insert, and delete)*

  - *Thus, Θ(nm)*

  - If we solve editor distance in a naïve D&C manner, what is the complexity?
    - *Exponential runtime.*
    - See Moodle for full computations.

DP Algorithm, bottom-up

```
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]
```

Trivial sub-problems i deletes and j inserts

Fills in entries in the (m+1)(n+1) matrix in row-major order.*

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

s="GO" t="LOG"

- m=2, n=3, we have a 3*4 matrix to fill in.

i \ j	0	1 L	2 O	3 G
0	0	1	2	3
1 G	1	1	2	2
2 O	2	2	1	2

```
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]
```

Trivial sub-problems i deletes and j inserts

Fills in entries in the (m+1)(n+1) matrix in row-major order.*

Remember edit operations

- m=2, n=3, we have a 3*4 matrix to fill in.

	0	1 L	2 O	3 G
0	0	1 I	2 I	3 I
1 G	1 D	1 R	2 R, I	2 C
2 O	2 D	2 R, D	1 C	2 I

```
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]
```

Trivial sub-problems
i deletes and j inserts

Fills in entries in the (m+1)(n+1) matrix in row-major order.*

Remember edit operations

- m=2, n=3, we have a 3*4 matrix to fill in.

	0	1 L	2 O	3 G
0	0	1 I	2 I	3 I
1 G	1 D	1 R	2 R, I	2 C
2 O	2 D	2 R, D	1 C	2 I

GO

LO

LO

LOG

i=1 Replace “L”

i=2 Copy “O”

i=2 Insert “G”

What is the running time of this algorithm? $\Theta(nm)$

Mini quiz

- s="GO" t="LOGG"

- Fill in the 3*5 matrix.
- Identify the edit distance and the corresponding edit operations.

- There may be more than one possible sequences of operations.

ILO of Lecture 1

- Dynamic Programming
  - To understand the principles of dynamic programming.
    - Overlapping sub-problems and optimal sub-structure.
    - Top-down with memoization and bottom-up.
  - To understand the DP algorithm for edit distance.
  - To be able to apply the DP algorithm design technique.

Lecture 2

- All-pairs shortest paths (dynamic programming)
  - To understand the adjacency matrix and the predecessor matrix, which are the representations of the input and output of most of the all-pairs shortest-path algorithms.
  - To understand how the dynamic programming principles play out in the Floyd-Warshall algorithm.