Algorithms and Data Structures

C8. Concepts

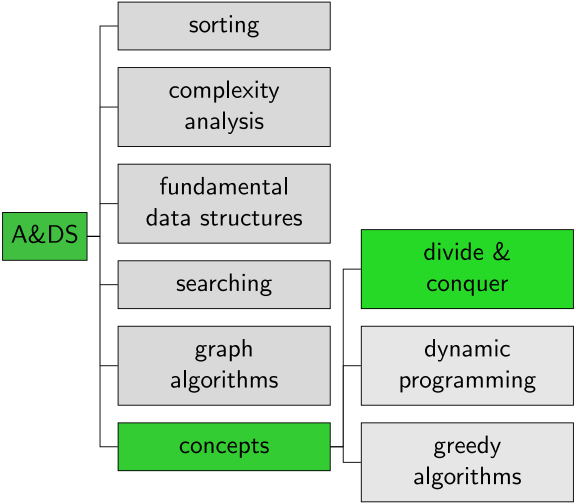
Gabriele Röger

University of Basel

May 30, 2024

C8.1 Divide and Conquer

Content of the Course



Recap: Divide-and-Conquer Algorithm Scheme

Base case: If the problem is small enough, solve it directly.

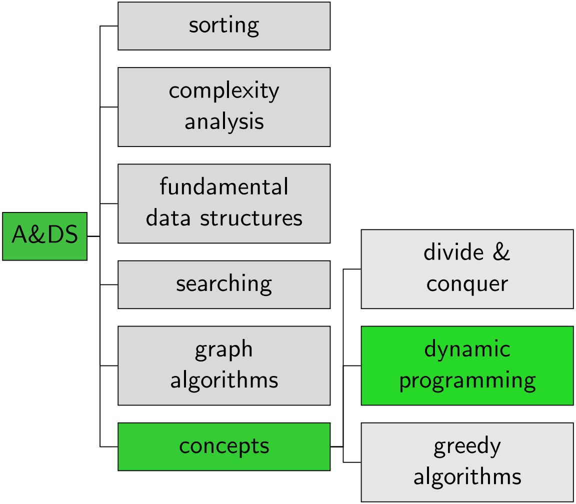
Recursive case: Otherwise

- **Divide** the problem into disjoint subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
- **Combine** the subproblem solutions to form a solution to the original problem.

Examples: Strassen's algorithm for multiplying square matrices, merge sort

C8.2 Dynamic Programming

Content of the Course



Dynamic Programming

Dynamic programming solves a problem by solving overlapping subproblems and combining their solutions.

Requirements:

- optimal substructure: (optimal) solutions of subproblems can be combined to (optimal) solutions of original problem
- overlapping subproblems: solving the subproblems requires solving common subsubproblems.

Solve each subsubproblem only once and store its solution.

Two Variants

- Top-down: Recursively call the algorithm for subproblems. If there already is a stored solution for the subproblem, use it. Otherwise solve it (recursively) and memoize its solution.
- Bottom-up: Solve the smallest subproblems first and combine their solutions into solutions of larger and larger subproblems.

Example: Fibonacci Numbers

The n-th Fibonacci number is

$$Fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{otherwise.} \end{cases}$$

We want to compute the n-th Fibonacci number.

Naive Implementation

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Exponential running time!

Dynamic Programming: Top-Down Variant

```
values = {0 : 0, 1 : 1}

def fibonacci(n):
    if n not in values:
        values[n] = fibonacci(n-1) + fibonacci(n-2)
    return values[n]
```

Linear running time

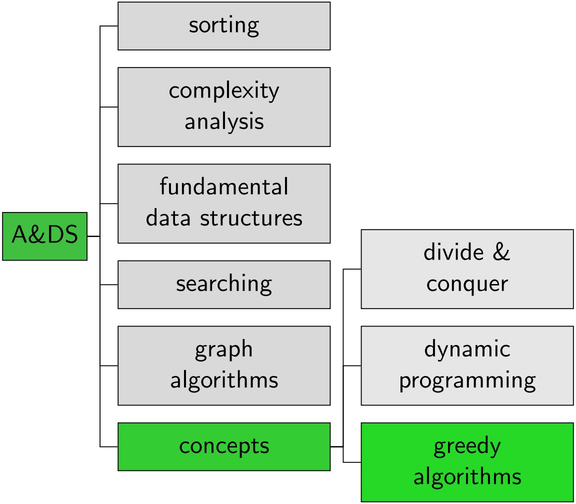
Dynamic Programming: Bottom-up Variant

```
def fibonacci(n):
    if n <= 1:
        return n
    prev_fib = 0
    curr_fib = 1
    for i in range(2, n+1):
        next_fib = prev_fib + curr_fib
        prev_fib = curr_fib
        curr_fib = next_fib
    return curr_fib
```

Linear running time

C8.3 Greedy Algorithms

Content of the Course



Greedy Algorithms

- A greedy algorithm always makes the choice that looks best at the moment (locally optimal choice).
- Some problems can be solved optimally with a greedy algorithm, but in general they lead to suboptimal solutions.

Example: Prim's Algorithm for Minimum Spanning Trees

Prim's Algorithm

- Choose a random node as initial tree.
- Let the tree grow by one additional edge in each step.
- Always add an edge of minimal weight
  that has exactly one end point in the tree.
  → locally optimal choice of edge
- Stop after adding $|V| - 1$ edges.

Knapsack Problem

- A burglar wants to steal items from a house and can carry at most K kilos.
- There are n items, where the ith items is worth v_i CHF and weights w_i kilos.
- The burglar wants to maximize the value of the stolen items.

Knapsack Problem: Greedy Strategy

- Greedy strategy: grab the items with the highest value per weight v_i/w_i as long as the total weight does not exceed K.
- Not guaranteed to lead to an optimal solution
  e.g. $K = 30, w_1 = 20, v_1 = 20, w_2 = w_3 = 15. v_2 = v_3 = 12$

Variant: Fractional Knapsack Problem

- In the fractional variant, the burglar can take away fractional amounts of an item.
  Think of the items as bags of gold dust.
- Greedy strategy: grab the items with the highest value per weight v_i/w_i as long as the total weight does not exceed K.
  If at the end there is room for a fraction of the next best item, take that fraction.
- Greedy strategy solves the problem optimally.