Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik, Anja Rey

Example solution sketches

Exercise Sheet 1

for the lecture on

Advanced Programming and Algorithms – Part II

This exercise sheet contains exercises for self-study and discussion.
If you would like feedback on your solutions, please upload a PDF via ILIAS until Monday, 15th April, 12:30 pm.

Submissions are no longer required to pass this course.

Discussion in the exercise classes from 15th April until 19th April, 2024.

Problem 1 to hand in: *Recursive Algorithms*

The following Python code and pseudocode contains errors. For each subtask determine an error and write down how it can be repaired.

why is it wrong? note if this is missing

Choose one subtask and prove formally that the repaired version is now correct.

a) Given a non-negative integer n, the following code should compute the power 3^n recursively.

no return value (recursion terminates, but no information is transferred)

```
def power_of_three(n):
    if n == 0:
        return 1
    power_of_three(n - 1) * 3
```

return

b) Given a list a that contains n integers, the following code should compute the index of the minimum entry in a recursively.

always returns 0

relation to original index is lost

```
def minimum_index(a, n):
    if n == 1:
        return 0
    min = minimum_index(a[1:n], n - 1)
    if a[min] < a[0]:
        return min
    else:
        return 0
```

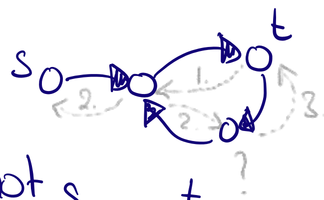
different solutions:
- min + 1
- return left and right indices
- split at the end a[0..n-1], n

c) Given a directed graph $G = (V, E)$, a source vertex $s \in V$, and a target vertex $t \in V$, the following code should compute the number of different paths from s to t.

number_of_paths($G = (V, E)$, s, t):

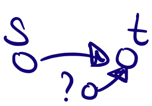
if $t = s$ **then**
  return 1
return $\sum_{(u,t)\in E}$ number_of_paths(G, s, u)

issue 1: graph could contain a cycle



recursive dependence

issue 2: even if cycle-free, there is not always a base case



↳ add if indegree(t) == 0 : return 0.

→ for self-study. questions? discussions?

Problem 2 as a programming exercise: *Depth First Search*

a) Implement a recursive version and an iterative version of Depth First Search in Python. Compare the two versions – what are their advantages and disadvantages?

   Hint: Before you look it up, think about how you would solve this yourself.

b) Take a closer look into how Depth First Search is implemented within `networkx`. What differences are there and when would you use what?

c) Implement an algorithm that returns a topological order of the vertices of a given acyclic graph.

→ lowest priority — also possible later

Problem 3 for discussion: *Acyclic Graph*

Design an algorithm that determines whether a given undirected graph $G = (V, E)$ contains a cycle. How can this be achieved with a running time in $\mathcal{O}(|V|)$ (indepented of the number of edges).

approaches for exercise / hints

- how in a directed graph without time constraint?
- what is different in undirected graph?
  - what do we know about cycle-free undir. graphs?
- which graph data structure do we need?

sketch:

- graph given as an adjacency list
- count edges. If there are more than $|V|-1$ edges $\Rightarrow$ a cycle exists
  - in $\mathcal{O}(|V|)$
    either number known,
    or stop counting at $|V|$
- otherwise perform DFS / top sort to find a cycle
  - in $\mathcal{O}(|V|+|E|) = \mathcal{O}(|V|)$ (since $|E| \leq |V|$)

→ (also possible later if necessary) medium priority context before mid May

Problem 4 for discussion: *Recursive vs Iterative Algorithms*

Take a closer look at the three (repaired) functions from Problem 1. How can they be implemented iteratively? What are their respective running times? What are other advantages and disadvantages of recursive functions?

a) ▷ loop with n iterations

▷ both in $O(n)$

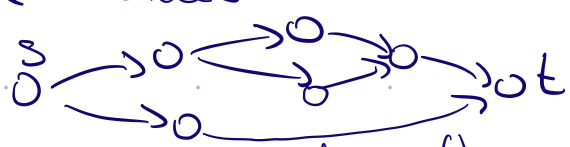
b) ▷ linearly run through all entries (have to see each entry at least once)

▷ both in $O(n)$ ← intuition for recursion, too

recursion: $T(n) = \begin{cases} \text{const.} & n = 1 \\ T(n-1) + \text{const.} & n > 1 \end{cases}$

⇝ closed form $T(n) = \text{const.} \cdot n$

c) ▷ How can we sort the vertices to compute values based on known values



⇝ topological order (known from lecture 1 and 2. c)

Dyn. programming not known, yet

⇝ intuition here already that values might be computed several times (ref. lecture 6 – 13th May)

Priority: spend at least 5 min on this to provide an idea $\rightsquigarrow$ rest also possible next week.

Problem 5 for discussion: *Solving Recurrences*

In the context of running times of recursive algorithms, we will come across recurrences of the form

$$T(n) = \begin{cases} c & n = 1 \\ a \cdot T(\frac{n}{b}) + f(n) & n > 1. \end{cases}$$

How can we solve them to obtain a closed form for $T(n)$ and classify them asymptotically?

Examples:

a) $T(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot T(\frac{n}{2}) + 1 & n > 1. \end{cases}$ assume $n = 2^k, k \in \mathbb{N}$ (general case $\rightsquigarrow$ Exercise 2)

b) $T(n) = \begin{cases} 1 & n = 1 \\ 9 \cdot T(\frac{n}{3}) + n^2 & n > 1. \end{cases}$ $n = 3^k, k \in \mathbb{N}$

c) $T(n) = \begin{cases} 1 & n = 1 \\ 16 \cdot T(\frac{n}{2}) + n^3 & n > 1. \end{cases}$ $n = 2^k, k \in \mathbb{N}$

d) $T(n) = \begin{cases} 1 & n = 1 \\ 16 \cdot T(\frac{n}{4}) + n^3 & n > 1. \end{cases}$ $n = 2^{2k}, k \in \mathbb{N}$

e) $T(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot T(\sqrt{n}) + \log n & n > 1. \end{cases}$

Recurrences: different techniques

- insert $\rightsquigarrow$ guess/observe structure & prove inductively
- tree (different degrees of precision
- use telescoping sum $\rightsquigarrow$ lecture 2
- "master thm" in lecture 3 (Mon. 22nd)

Options for exercise: • split up into groups (if large enough) + present one technique each

OR • choose individually which new techniques to tackle $\rightsquigarrow$ leave open for self-study + questions next week examples:

a) guess $T(n) \in O(n)$. correct, but showing $T(n) \le c \cdot n$ doesn't work straightforwardly.

$\rightsquigarrow$ new hypothesis: $T(n) \le c \cdot n - d$ $\quad (d \ge 0$ const.)

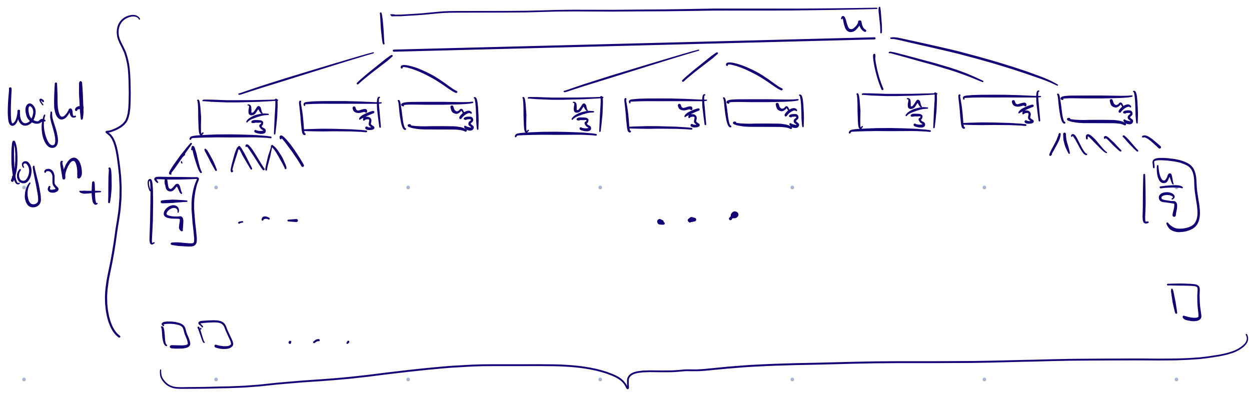
induction: • $T(1) = 1 \le c - d$ for $c - d \ge 1$

• $1, \dots, n-1 \rightsquigarrow n$: $T(n) = 2 \cdot T(\frac{n}{2}) + 1 = 2 \cdot (c \cdot \frac{n}{2} - d) + 1$

$= c \cdot n - 2d + 1 \le c \cdot n - d$ for $\underline{d \ge 1}$

caution: has to be the same constant (see also lecture example)

b) $T(n) = \begin{cases} 1 \\ 9 \cdot T(\frac{n}{3}) + n^2 \end{cases}$



n^2

$9 \cdot (\frac{n}{3})^2 = n^2$

$9^2 \cdot (\frac{n}{3^2})^2 = n^2$

$\vdots$

n^2

$(\log_3 n + 1) \cdot n^2$

$\in \Theta(\log_3 n \cdot n^2)$

(„case 2")

$9^{\log_3 n} = n^{\log_3 9} = n^2$

$\Big\lceil a^{\log_b n} = \underbrace{(b^{\log_b a})}_{a}{}^{\log_b n} = \underbrace{(b^{\log_b n})}_{n}{}^{\log_b a} = n^{\log_b a} \Big\rfloor$

c) $T(n) = \begin{cases} 1 \\ 16 \cdot T(\frac{n}{2}) + n^3 \end{cases}$ (case: $16^{\log_2 n} = n^4 \rightsquigarrow \Theta(n^4)$)

telescoping sum: $S_i := \frac{T(2^i)}{16^i}$

$$\sum_{i=1}^{\log n} (S_i - S_{i-1}) = \sum_{i=1}^{\log n} \left(\frac{T(2^i)}{16^i} - \frac{T(2^{i-1})}{16^{i-1}} \right) = \sum_{i=1}^{\log n} \left(\frac{\cancel{16 \cdot T(2^{i-1})} + 2^{3i}}{16^i} - \frac{\cancel{16 \cdot T(2^{i-1})}}{16 \cdot 16^{i-1}} \right) = \sum_{i=1}^{\log n} \frac{2^{3i}}{2^{4i}}$$

$$= \sum_{i=1}^{\log n} \frac{1}{2^i} \underset{\text{geom. series}}{=} \frac{1 - (\frac{1}{2})^{\log n + 1}}{1 - \frac{1}{2}} = \frac{1 - \frac{1}{2} \cdot \frac{1}{n}}{\frac{1}{2}} = 2 - \frac{1}{n}$$

// telesc.

$S_{\log n} - S_0$

$$= \frac{T(2^{\log n})}{16^{\log n}} - T(1) = \frac{T(n)}{n^4} - 1$$

equal

$\Leftrightarrow T(n) - 1 \cdot n^4 = 2 \cdot n^4 - \frac{n^4}{n}$

$\Leftrightarrow T(n) = 3n^4 - n^3 \in \Theta(n^4)$

d) $T(n) = \begin{cases} 1 \\ 16 \cdot T(\frac{n}{4}) + n^3 \end{cases}$ case $16^{\log_4 n} = n^2 \rightsquigarrow \Theta(n^3)$

e) $T(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot T(\sqrt{n}) + \log n & n > 1 \end{cases}$

trick: substitute to something known.

renaming $m = \log n$

$T(2^m) = \begin{cases} 1 & m = 0 \\ 2 \cdot T(2^{m/2}) + m & m > 0 \end{cases}$

$S(m) := T(2^m) \Rightarrow S(\frac{m}{2}) = T(2^{m/2})$

$\rightsquigarrow S(m) = \begin{cases} 1 \\ 2 \cdot S(\frac{m}{2}) + m \end{cases}$ known to be in $\Theta(m \log m)$ (e.g. merge sort)

$\Rightarrow T(n) = T(2^m) = S(m) \in \Theta(\log n \log \log n)$.