Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik, Anja Rey

Example Solution sketches

Exercise Sheet 2

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, 22nd April, 12:30 pm.

Discussion in the exercise classes from 22nd April until 26th April, 2024.

Problem 1 to hand in: *Analyse a Divide & Conquer Algorithm* (from former mock exam)

Consider the following algorithm with an input array A consisting of integers, sorted ascendingly, and two indices ℓ and r, $0 \leq \ell \leq r \leq \text{length}(A) - 1$.

ℓ r n $\quad$ $n = r - \ell + 1$

do_something(A, ℓ, r): $\quad$ running time $T(n)$

- **if** $\ell = r$ **then** $\quad$ const. $\}$ const. (base case)
  - **return** ∞
- $m \leftarrow \lfloor \frac{r+\ell}{2} \rfloor$ $\quad$ const
- $a \leftarrow$ do_something(A, ℓ, m) $\quad$ $T(\frac{n}{2})$
- $b \leftarrow$ do_something($A, m+1, r$) $\quad$ $T(\frac{n}{2})$
- $c \leftarrow |A[m+1] - A[m]|$ $\quad$ const
- **return** $\min\{a, b, c\}$ $\quad$ const

a) Given a sorted array A, and, initially, $\ell = 0$ and $r = \text{length}(A) - 1$, briefly explain what the algorithm do_something does and what it returns.

b) Let the input size $n = r - l + 1$ be a power of two ($n = 2^k$ for some $k \in \mathbb{N}$). Analyse the asymptotic worst-case running time $T(n)$ of do_something:

- Write down which parts of the algorithm take which running time (up to constants). These can depend on the running times $T(n')$ of further calls of the algorithm with a smaller input n'.
- Provide a recurrence that describes $T(n)$ for each $n \geq 1$.
- Solve the recurrence to obtain a closed formula as an (ideally precise) upper bound for $T(n)$ which is the running time for an array of input length n.
- Provide a function as a representative asymptotic upper bound in $\mathcal{O}$-notation.

c) Provide a sketch of a correctness proof for this algorithm do_something.

d) In the algorithm above, we assume that the input array is sorted ascendingly. If this is not the case, how can we sort the array first? Briefly explain how it works and how this effects the overall running time of the algorithm.

a) divide & conquer approach:
do_something recursively calls itself on two halves of the input array and returns the minimum of the two recursively returned values and the distance between the largest array entries of the first half and the smallest entry of the second half.
It terminates if the current subarray length is only 1 and returns ∞.
This way it computes the smallest distance between two entries of the initial array.

b) from line-by-line notes above:

$$T(n) = \begin{cases} \text{const} & n = 1 \\ 2 \cdot T(\frac{n}{2}) + \text{const} & n > 1,\ n = 2^k \end{cases}$$

$\rightsquigarrow$ e.g. via Master Thm

$$T(n) \in O(n)$$

different ways to solve this

c) claim: given A, l, r, do_something returns the min distance between two entries in $A[l..r]$ (or ∞ if there's only one entry)

proof: induction over input size $n = r - l + 1$:

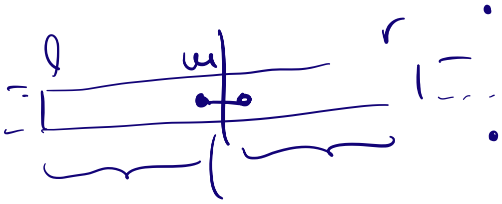
base case: $n = 1 \Leftrightarrow r = l$

l. 1 and 2 $\rightsquigarrow$ return ∞

(extended) ind. hypothesis $\forall\ k,\ 1 \le k < n$

$\rightsquigarrow$ step: show that

- a = min dist. within $\underbrace{A[l..m]}_{\text{hyp holds, } m-l+1<n}$
- b = min dist within $\underbrace{A[m+1..r]}_{r-m-1+1<n}$



- the smallest distance can only occur between two neighbouring entries.
  The only pair not considered, yet, is between the two subarrays $A[m], A[m+1]$
- l.7: min of the three is returned.

d) sort, e.g., with Merge Sort in the beginning
- divide & conquer approach:
- cut in half, solve each half recursively, terminates if only one element left
- merge two parts linearly to sort
- in $O(n \log n)$ time

Problem 2 for discussion: *Binary Search*

Let A be an array consisting of $n \geq 1$ distinct integers, sorted in descending order.

Design an algorithm that, given A and an integer x, returns index i if $A[i] = x$ and -1 if A does not contain x. The algorithm should work by the divide and conquer principle and run in logarithmic time $\mathcal{O}(\log n)$ in the input size n.

Optionally, you can use additional in- and output parameters to store indices or other temporary values.

For instance, given the following array and $x = 11$, the algorithm should return index 2.

0	1	2	3	4	5	6
17	13	11	9	7	5	3

Proceed as follows:

a) Describe the algorithm intuitively and provide pseudocode.

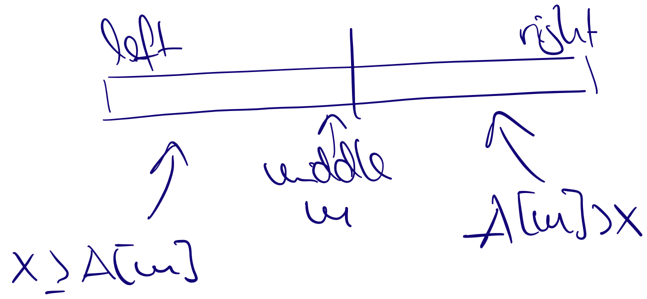
b) Argue why the running time of $\mathcal{O}(\log n)$ is fulfilled.

c) Argue why it works.

a) Binary Search:

Task: think about solution and analysis (before looking it up) consider: divide & conquer approach:

!
- what is the base case?
- how do I divide the array? parameters?
- what do I do to combine the information from deeper recursive levels?
  ↳ where do I continue the search
- how many recursive calls do I need?



if $A[m] > x$
$\rightsquigarrow$ search in $A[m+1..r]$
if $A[m] \leq x$ $\rightsquigarrow A[l\_m]$
(or if $=$, m already found)

init $l = 0$, $r = n-1$

Binary Search (A, l, r, x) $\qquad \frac{T(n)}{}$
  if $l = r$: $\qquad c$
    if $A[l] = x$: return l
    else: return -1
  $m = \lfloor \frac{l+r}{2} \rfloor$ $\qquad c$
  if $A[m] > x$:
    return Binary Search$(A, m+1, r, x)$ $\qquad T(\frac{n}{2})$
  else: return BinarySearch(A, l, m, x) $\qquad$ OR $T(\frac{n}{2})$

b) $T(n) = \begin{cases} c & n = 1 \\ T(n) + c & n > 1 \end{cases}$ $\rightsquigarrow$ $T(n) \in \mathcal{O}(\log n)$

$n = r - l + 1$

c) induction over $n = r - l + 1$:
if $x \in A[l..r]$ Binary Search returns i with $A[i] = x$
(and -1 otherwise)

Problem 3 as a programming exercise: *Merge Function*

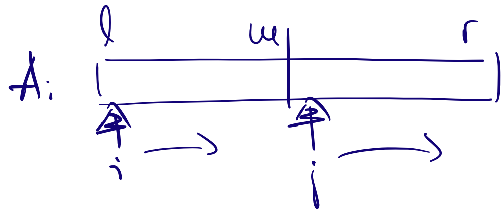
Implement the `merge` function as required for merge sort: Given an array A and three indices, `left`, `middle`, and `right` such that $A[$`left..middle`$]$ and $A[$`middle`$+1..$`right`$]$ are sorted ascendingly, return A with $A[$`left..right`$]$ sorted ascendingly.

At some point the algorithm should iterate over (a part of) the array. What would a loop invariant look like that can be used to show the correctness of the algorithm?

What would a unit test look like that asserts this property exemplarily?

How efficient is your implementation? What changes if we use different data structures?

array of length $r-l+1$ needed to either back up the original parts (book) or store the new part (here).



merge(A, l, m, r):

- $B \leftarrow$ new array $[0 .. r-l]$
- $i \leftarrow l$
- $j \leftarrow m+1$
- for $k \leftarrow 0$ to $r-l$: ($\hat{=}$ range$(r-l+1)$, k index for B)
  - if $i \leq m$ and $(A[i] \leq A[j]$ or $j > r)$: smaller value chosen; right part empty
    - $B[k] \leftarrow A[i]$
    - $i \leftarrow i+1$
  - else: ($\hat{=}$ left part empty or larger value)
    - $B[k] \leftarrow A[j]$
    - $j \leftarrow j+1$
- $A[l .. r] \leftarrow B[0 .. r-l]$

refactor?

loop invariant: before iteration with k ($k+1^{st}$ iteration): $B[0..k-1]$ k smallest elem. among $A[l..r]$ sorted and next element ($k+1^{st}$ smallest in $A[i]$ or $A[j]$)

(lower priority.)

Problem 4 for discussion: *Recurrences – General Case*

For the sake of simplicity, we assumed in several instances that an input size n is a power of b if a recursion depends on a subproblem of size n/b.

Why can we make this assumption without loss of generality?

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

similar for $\lceil \frac{n}{b} \rceil$

what if $n \neq b^k, k \in \mathbb{N}$? $\rightsquigarrow$ $\exists k \in \mathbb{N}: \quad b^k < n < b^{k+1}$

assume: $f(n)$ (weakly) incr. monotonic (reasonable for running time)

discussion:

steps: $\triangleright$ show (weak) monotonicity of $T(n)$:

$$T(b^k) \overset{!}{\leq} T(n) \overset{!}{\leq} T(b^{k+1}):$$

$$T(b^k) = a \cdot T(b^{k-1}) + f(b^k) \leq a \cdot T(\lfloor \tfrac{n}{b} \rfloor) + f(n) = T(n)$$
$$\leq a \cdot T(\tfrac{b^{k+1}}{b}) + f(b^{k+1}) = T(b^{k+1})$$

$\triangleright$ show that if $T(b^k) \leq c \cdot g(b^k)$ and $T(b^{k+1}) \leq c \cdot g(b^{k+1})$ for some mon. incr. fct. g, $f(n) \in g(n)$ (compare cases in Thm ...)

then $T(n) \leq c' \cdot g(n)$

(some constant $c' > c$ $\forall n \geq n_0$ ..)

$$T(n) \leq T(b^{k+1}) \leq c \cdot g(b^{k+1})$$

$$a \cdot T(b^k) + f(n) \leq a \cdot c \cdot g(b^k) + c'' \cdot g(n) \leq (ac + c'') \cdot g(n)$$

($\forall n \geq n_0$ ..)