Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik, Anja Rey

Example Solution Sketches

Exercise Sheet 3

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 29th April until 3rd May, 2024.

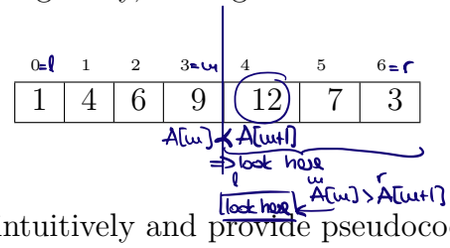
Problem 1 to hand in: *Design a Divide & Conquer Algorithm*

Let A be an array consisting of $n \geq 1$ distinct integers, sorted partly in ascending, and partly in descending order. More precisely, there exists some peak index p, $0 \leq p \leq n-1$, such that $A[i] < A[i+1]$, for $0 \leq i \leq p-1$, and $A[i] > A[i+1]$, for $p \leq i \leq n-2$. The exact position p is not known in advance.

Design an algorithm that, given A, returns the peak index p. The algorithm should work by the divide and conquer principle and run in linear time $\mathcal{O}(\log n)$.

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

For instance, given the following array, the algorithm should return index 4.



Proceed as follows:

a) Describe the algorithm intuitively and provide pseudocode.

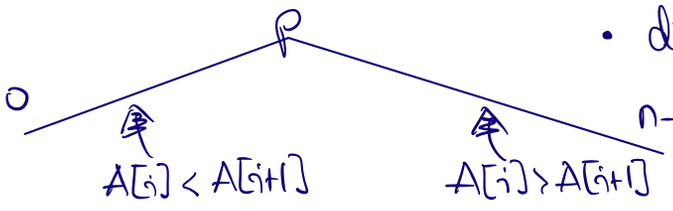
b) Analyse why the running time of $\mathcal{O}(\log n)$ is fulfilled.

c) Briefly argue why it works?

---

a) Idea: DAC approach:

- parameters left l and right r for subarray (init l=0, r=n-1)
- divide into two parts ($m = \lfloor \frac{l+r}{2} \rfloor$)
- if $A[m] < A[m+1]$ ⇒ p is located in $A[m+1..r]$
- otherwise p in $A[l..m]$
- if $l = r$, p is found. return l.



b) analysis similar to BinarySearch:
→ ref to algorithm: one recursive call, combination in $\mathcal{O}(1)$
→ $T(n) = \begin{cases} c & n=1 \\ 1 \cdot T(\frac{n}{2}) + c & n>1 \end{cases}$ $(c>0)$ → ⇒ $T(n) \in \mathcal{O}(\log n)$

c) Claim: if the current subarray is structured as intended, then the algorithm returns the peak index.

this can be shown via induction over $n = r - l + 1$.

key arguments:
- the structure is maintained in the recursive call
- the subarray in the recursive call contains p.

Problem 2 for discussion: *Longest Increasing Contiguous Subsequence*

Let A be an array consisting of $n \geq 1$ distinct integers. A contiguous subsequence of $A[i..j]$ is increasing if $A[k] < A[k+1]$ for each $i \leq k < j$. The length of this subsequence is $j - i + 1$. For instance, given the following array, the length of the longest increasing contiguous subsequence is 3.

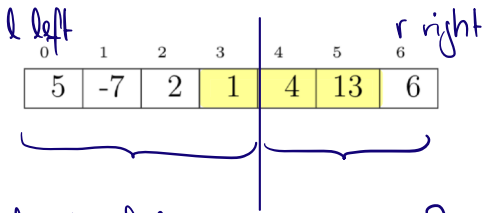
0	1	2	3	4	5	6
5	-7	2	1	4	13	6

Design an algorithm that, given A, returns the length of the longest increasing contiguous subsequence within A. The algorithm should work by the divide and conquer principle and run in logarithmic time $\mathcal{O}(n)$.

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

▷ which cases can occur?

▷ How can we combine the information?



e.g. 3 return values:
- $x_l = 1$ length of l.i.c.s. starting in l — $y_l = 2$
- $x_r = 1$ " ending in r — $y_r = 1$
- $x = 2$ " overall within subarray — $y = 2$

combine:

$z = \max\{x, y, x_r + y_l\}$ if $A[m] < A[m+1]$

$z_l = \begin{cases} x_l + y_l & \text{if } x_l = \lceil \frac{n}{2} \rceil \text{ and } \dots \\ x_l & \text{otherwise} \end{cases}$

$z_r = \begin{cases} x_r + y_r & \text{if } y_r = \lfloor \frac{n}{2} \rfloor \text{ and } \dots \\ y_r & \text{otherwise.} \end{cases}$

in constant time

$T(n) = \begin{cases} 2\,T(\frac{n}{2}) + c \\ c \end{cases}$

$\in \mathcal{O}(n)$.

Problem 3 as a programming exercise: *Natural Merge Sort*

Consider an algorithm that combines the advantages of insertion sort and merge sort. Instead of breaking the input array down into subarrays of length 1, we want to break it down to runs of already sorted subarrays. The algorithm should work as follows: Find the runs. Split the array into two parts with an equal number of runs. Recursively, perform the algorithm on both parts until there is only one run left. Merge two parts with the known function from Merge Sort.

a) Implement the algorithm. In particular, think about how index parameters for both, the array and the runs, are managed?

b) How can your implementation be refactored?

c) What is the worst-case running time?

d) What is the best-case running time?

e) Which sorting algorithm is used for the built-in `list.sort()` and `sorted()` functions in Python? What are their advantages?

a)

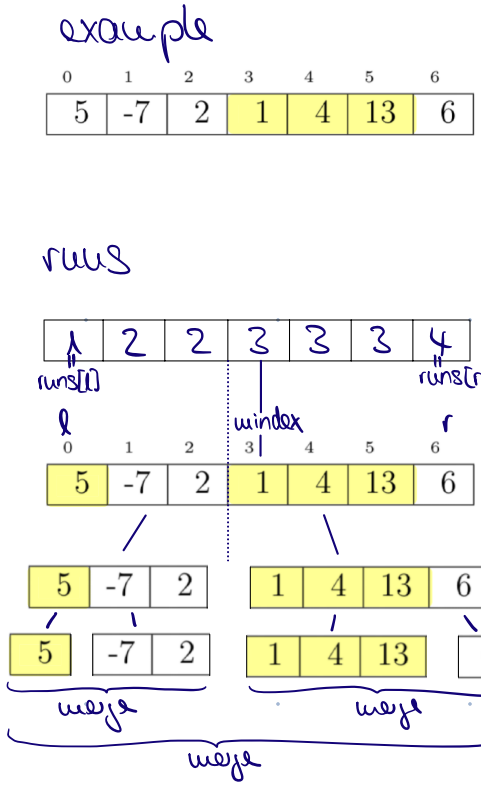
```
natural_sort(A):
 1 runs ← find_runs(A)
 2 natural_merge_sort(A, runs, 0, len(A))

find_runs(A):
 1 runs ← array of length len(A)
 2 r ← 1
 3 runs[0] ← r
 4 for i ← 1 to len(A) − 1 do
 5 |   if A[i − 1] > A[i] then
 6 |   |   r ← r + 1
 7 |   runs[i] ← r
 8 return runs

natural_merge_sort(A, runs, ℓ, r):
 1 if runs[ℓ] ≠ runs[r] then
 2 |   mrun ← ⌊(runs[ℓ]+runs[r])/2⌋
 3 |   mindex ← search_first(runs, m + 1)
 4 |   natural_merge_sort(A, runs, ℓ, mindex − 1)
 5 |   natural_merge_sort(A, runs, mindex, r)
 6 |   merge(A, ℓ, mindex − 1, r)
```

array to manage

similar to merge sort with runs as a base



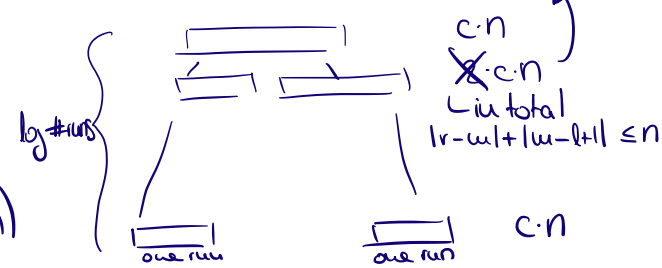
b) ⇒ subfunctions, rename, ... ⇒ work on actual implementations

c) worst case : find_runs in $O(n)$

natural_merge_sort : $T(\#runs) = \begin{cases} c & \#runs = 1 \\ 2 \cdot T(\frac{\#runs}{2}) + c \cdot n \end{cases}$

(decreasing order $\#runs = n$)

(alg. $\#runs \le n$ " $r-l+1$)



in total $O(\log(\#runs) \cdot n) \le O(\log n \cdot n)$

d) best case : sorted in increasing order :

only one run

=> • find_runs still in $O(n)$

• natural_merge_sort base case, already sorted

in $O(n)$

e) "Tim-Sort" 2002 similar combination of merge sort with natural runs

based on idea that data is often partially sorted.

[wiki.python.org]

additional balance criteria

Problem 4 for discussion: *Bucket Sort*

In Lecture 3 (see slide *Sorting Algorithms Overview*), sorting algorithms are mentioned that run in linear time. If the input array consists of integert from a fixed finite set M of size $|M| = m$, we can consider the following approaches.

a) Given an array A with n integers from M, *Bucket Sort* works as follows: Initialise an array B of size m (with a placeholder not in M), the *buckets*. For each element x in A, write x into the corresponding bucket $B[x]$. Then, iterate over B to copy all numbers (except the placeholder) back to A.

   Implement the details of this approach. How can multiple occurrences be considered? Show that it works in a running time in $\mathcal{O}(m + n)$ and space $\mathcal{O}(m + n)$.

b) Let d be the maximal number of digits of an integer in M. Given an array A with n integers from M, *Radix Sort* works as follows: Initialise an array B of d buckets (e.g., lists). For each digit i, $0 \leq i \leq d$, (representing 10^i), sort all integers from A into $B[x]$ wrt their ith digit x. Copy all integers from B in order back to A.

   Implement the details of this approach. Why does this work? Show that it works in a running time in $\mathcal{O}(b + dn)$ and space $\mathcal{O}(b + n)$.

c) Why is the linear running time no contradiction to the lower bound of $\Omega(n \log n)$ (see slide *Sorting: A Lower Bound*)?