Algorithm Theory, Tutorial 1

~~Improving the QLever Search Engine~~

Johannes Kalmbach

University of Freiburg

johannes.kalmbach@gmail.com

October 2018

General Hints

- Contact tutor (`johannes.kalmbach@gmail.com`) for questions concerning corrections etc.
- Contact forum (`daphne.informatik.uni-freiburg.de`) for everything else
- Suggestion: Submit in groups of two (better for understandable algorithms)
- Submit readable solutions (LaTeX as pdf, CLEAN handwriting (+ good scan if necessary))
- Spend enough time on exercise sheets and writeup (you and I have to understand your submission).

Algorithm Writeups

- Pseudocode, limit to important aspects
- Reader must be able to understand and implement it.
- E.g "Split Array A in two evenly-sized halves L and R"

Ex 1a: Prove or disprove: $n! \in \Omega(n^2)$

Choose $n_0 = 2$, $c = 0.5$

$n \geq n_0 = 2$

$$n! = n \cdot (n-1) \cdot (n-2)! \geq n \cdot (n-1) = \underbrace{n^2 - n \overset{n \geq 2}{\geq} 0.5 \cdot n^2}$$

$$n^2 - n \geq 0.5 n^2$$
$$\Leftrightarrow 1 - \frac{1}{n} \geq 0.5$$

True for $n \geq 2$

Ex 1b: Prove or disprove: $\sqrt{n^3} \in O(n \log n)$

Claim $\exists c, n_0 : \ldots$

$\forall n \geq n_0$

$\sqrt{n^3} \leq c \cdot n \cdot \log n$

for large enough n

$\sqrt{n} \leq c \cdot \log n \leq n^{\frac{1}{4}} \cdot c$

Hint

$: n^{\frac{1}{2}}$

$\Rightarrow 1 \leq n^{-\frac{1}{4}} \cdot c'$ constant $\Rightarrow \lightning$

$\rightarrow 0$

for large enough n

Ex 1b: Prove or disprove: $\sqrt{n^3} \in O(n \log n)$

Ex 1b: Prove or disprove: $\sqrt{n^3} \in O(n \log n)$

$\Rightarrow 2^{\sqrt{\log_2 n}} \in \Omega(n)$

$\Rightarrow \exists n_0, c, \forall n \geq n_0 \quad 2^{\sqrt{\log n}} \geq c \cdot n$

$\Rightarrow \quad \sqrt{\log n} \geq \log c + \log n$

$\Rightarrow \quad 1 \geq \frac{\log c}{\sqrt{\log n}} + \sqrt{\log n}$

$\frac{\log c}{\sqrt{\log n}} \to 0 \qquad \sqrt{\log n} \to \infty$

$\Rightarrow$ ↯ for large n

$\Rightarrow 2^{\sqrt{\log_2 n}} \notin \Omega(n)$

Ex 1c: Prove or disprove: $2^{\sqrt{\log_2 n}} \in \Theta(n)$

Ex 1c: Prove or disprove: $2^{\sqrt{\log_2 n}} \in \Theta(n)$

Sort by asymptotic growth

$$\sqrt{\log n} < \log(\sqrt{n}) = \log n = \log(n^2)$$
$$< \log^2(n) < 2^{\sqrt{\log_2 n}} < \sqrt{n} < n$$

$< \in o(\text{next})$

asymptotic

n^2	$\sqrt{n}$	$2^{\sqrt{n}}$	$\log(n^2)$
$2^{\sqrt{\log_2 n}}$	$\log(n!)$	$\log(\sqrt{n})$	$(\log n)^2$
$\log n$	$10^{100} n$	$n!$	$n \log n$
$2^n / n$	n^n	$\sqrt{\log n}$	n

$$= 10^{100} n < \underbrace{n \cdot \log n = \log(n!)} < n^2$$
$$< 2^{\sqrt{n}} < \frac{2^n}{n} < n! < n^n$$

Sort by asymptotic growth

$$n! = \underbrace{n \cdot (n-1) \cdot (\ldots) \cdot}_{\frac{n}{2} \text{ factors that are } \geq \frac{n}{2}} \left(\frac{n}{2}\right)! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

$$\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$$

$$\Rightarrow \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) \leq \log(n!) \leq n \cdot \log n$$

Sort by asymptotic growth

Ex 3: Master Theorem

See master solution, just table lookup

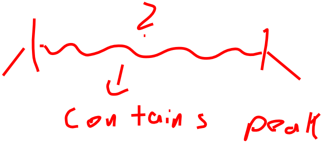
Ex 4: Peak Elements

You are given an array $A[1 \dots n]$ of n integers and the goal is to find a peak element, which is defined as an element in A that is equal to or bigger than its direct neighbors in the array. Formally, $A[i]$ is a peak element if $A[i-1] \leq A[i] \geq A[i+1]$. To simplify the definition of peak elements on the rims of A, we introduce *sentinal-elements* $A[0] = A[n+1] = -\infty$.

a. Give an algorithm with runtime $O(\log n)$ (measured in the number of read operations on the array) which returns the position i of a peak element.
b. Prove that your algorithm always returns a peak element, give a recurrence relation for the runtime and use it to prove the runtime.

Claim: If $A[0] < A[1]$ and $A[n] > A[n+1]$ then $A[1 \dots n]$ contains a peak element.

Prove by contradition: Assume that $A[1 \dots n]$ contains no peak, then



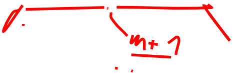
$$\Rightarrow A_1 < A_2 < A_3 < A_4 < \dots < A_{n-1} < A_n > A_{n+1}$$

else A_1 is peak.

similar

required.

Idea: Make the considered array smaller while ensuring that it still contains a peak element:




```
fun peak(Arr):
    m = middle index in Arr
    if Arr[m] is peak, return m
    if Arr[m-1] ≥ Arr[m]
        return peak(Arr[1..m-1])
else
    return peak(Arr[m..end])
```

Constructing the algorithm

Proof of correctness

Two parts: Algorithm always terminates and only looks at arrays that contain peak elements.

Beginning: peak exists (because of sentinels)

~~Re~~ if input has risings before and after

then smaller recursed array has risings

$\Rightarrow$ Base case:

Array of size 1, that has risings before + after $\Rightarrow$ peak

Proof of correctness

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

only recurse to one half

look at middle

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

Proof of correctness

Ex 5, Frequent Numbers

You are given an Array $A[0 \dots n-1]$ of n integers and the goal is to determine frequent numbers which occur at least $n/3$ times in A. There can be at most three such numbers, if any exist at all.

a) Give an algorithm with runtime $O(n \log n)$ (measured in number of array entries that are read) based on the divide and conquer principle that outputs the frequent numbers (if any exist).

b) Argue why your algorithm is correct, give a recurrence relation for the runtime and use it to prove the runtime.

First Try: HashMap

```
d = dict()          # Hash Table

for el in array
    d[el] += 1      # O(n) in
                    #   worst case
for el in dict:
    if dict[el] ≥ n/3
        output el
```

fun freq (Arr);
if len(Arr) ≤ 3): ... (trivial) O(1)
l = freq (left half)
r = freq (right half)

len(Arr)
$\downarrow$
$\geq \frac{n}{3}$

for K in l ∪ r:
If K occurs sufficiently often in Arr, add it to output
(Count candidates in full array)

Runtime

Recursion

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

$$\Rightarrow T(n) \in \Theta(n \cdot \log n)$$

1. Sort the array $(n \cdot \log n)$

$$\begin{array}{cccccccccccccc} 2 & & & & 3 & 1 & & 2 & & & & & & \\ 00 & 1 & 1 & 1 & 2 & 33 & 4 & 4 & 4 & 4 & \ldots \end{array}$$

$\downarrow$

2. Determine freq. els in linear time