Algorithmen & Datenstrukturen
Woche 3

Julian Steinmann

11. Oktober 2021

ETH Zürich

MAXSUBARRAYSUM

Ideen

- Precomputing (Präfixsummen)
- Divide-and-Conquer
- Sich Resultate merken

`i`	0	1	2	3	4	5
`array[i]`	1	6	-4	2	5	1

Wir wollen viele (z.B. q) Summen von Subarrays möglichst effizient berechnen.
Wie lange dauert die naive Version (in O-Notation)?

i	0	1	2	3	4	5
array[i]	1	6	-4	2	5	1

Wir wollen viele (z.B. q) Summen von Subarrays möglichst effizient berechnen.
Wie lange dauert die naive Version (in O-Notation)?

$$\rightarrow O(nq)$$

Precomputing (Präfixsummen)

`i`	0	1	2	3	4	5
`array[i]`	1	6	-4	2	5	1
`pre[i]`	1	7	3	5	10	11

Wie lange dauert es, `pre` zu berechnen?

Precomputing (Präfixsummen)

`i`	0	1	2	3	4	5
`array[i]`	1	6	-4	2	5	1
`pre[i]`	1	7	3	5	10	11

Wie lange dauert es, `pre` zu berechnen?

$$\rightarrow O(n)$$

Wie lange dauert es nun q Summen von Subarrays zu berechnen?

Precomputing (Präfixsummen)

`i`	0	1	2	3	4	5
`array[i]`	1	6	-4	2	5	1
`pre[i]`	1	7	3	5	10	11

Wie lange dauert es, `pre` zu berechnen?

$$\rightarrow O(n)$$

Wie lange dauert es nun q Summen von Subarrays zu berechnen?

$$\rightarrow O(n + q)$$

Probleme in Subprobleme aufzuteilen kann das Problem einfacher lösbar machen. Divide-and-Conquer ist an sehr vielen Stellen anzutreffen.

Bei vielen Divide-and-Conquer-Problemen entstehen Zwischenresultate, welche wir uns sinnvollerweise merken. Dadurch müssen wir sie nicht mehrmals berechnen, sondern können sie wiederverwenden. Wir werden dies später im Semester konkreter sehen (→ *Dynamische Programmierung*).

Wir schauen uns nur ein Subproblem an: Was ist das Subarray mit der grössten Summe, welches bei Position i aufhört?
Weshalb ist dies ein "gutes" Subproblem?

Wir schauen uns nur ein Subproblem an: Was ist das Subarray mit der grössten Summe, welches bei Position i aufhört?
Weshalb ist dies ein "gutes" Subproblem? → Gute Subprobleme sind einfacher lösbar und erlauben, die ursprüngliche Lösung wieder zusammenzusetzen.

MSS - Kadane's Algorithmus

```python
def max_subarray_sum(numbers):
    best_sum = 0
    current_sum = 0

    for x in numbers:
        current_sum = max(current_sum + x, 0)
        best_sum = max(best_sum, current_sum)

    return best_sum
```

O-Notation in der Praxis

Laufzeit

Für einen Input von $n = 5000$ erhalten wir folgende Resultate:

	Komplexität	Zeit in ms
Naiv	$O(n^3)$	52’600
Divide-and-Conquer	$O(n \log n)$	17.3
Kadane	$O(n)$	2.5

→ O-Notation nicht exakt, aber trotzdem aussagekräftig

(Implementation in Python, Resultate von David Zollikofer)