Maximum Sub Sequence Problem – Lösungen

Für die gestellte Aufgabe ergeben sich mehrere Lösungsmöglichkeiten, über die hier kurz eine Übersicht gegeben werden soll.

Programm 0 – «Pythonic»-Lösung

Die Sprache *Python* erlaubt Programme auf einer hohen Abstraktions-Stufe zu formulieren. So kann die in der Aufgabe angegebene Formel recht direkt in ein Python-Programm übersetzt werden:

$$result = max\left(\{0\} \cup \bigcup_{start=0}^{a.length-1} \bigcup_{end=start}^{a.length-1} \sum_{i=start}^{end} a[i]\right)$$

zu

```
max([0] + [sum(a[start:end + 1]) for start in range(len(a)) for end in range(start, len(a))])
```

Aufpassen muss man lediglich bei den verschiedenen Obergrenzen des Listen-*Slices* und der Bereiche für die Werte von `start` und `end`.[1]

Dieses Programm ist kurz und durch seine direkte Entsprechung zur Mathematik leicht verständlich, wenn man sich einmal daran gewöhnt hat, *List Comprehensions* zu lesen. Ausserdem verwendet es keine Schleifen im eigentlichen Sinne. Weiterhin sind `sum` und `max` *build-in*-Funktionen, von denen wir uns besondere Effizienz erhoffen.

Aber ist dieses Programm wirklich effizient?

Um diese Frage zu beantworten, muss man einen Blick hinter die Kulissen werfen und überlegen, wie sich die gestellte Aufgabe sonst noch lösen lässt.

Programm 1

Verwendet man anstelle der *Comprehension* zwei geschachtelte Schleifen kann der grösste Wert der Summen auch mit einer Fallunterscheidung gesucht werden. Die Zwischenspeicherung in einer Liste ist dann nicht notwendig:

```
max_sum = 0
for start in range(len(a)):
   for end in range(start, len(a)):
      cur_sum = sum(a[start:end + 1])
      if cur_sum > max_sum:
         max_sum = cur_sum
```

Dieses Programm übersetzt ebenfalls die zugrunde gelegte mathematische Formel und liesse sich bei Bedarf auch leicht erweitern, um neben der maximalen Summe auch Start- und End-Index der entsprechenden Sub-Liste zu ermitteln.

Um genauer zu verstehen, was bei der Ausführung dieses Programms passiert, kann man die `sum`-Funktion durch eine zusätzliche Schleife ersetzen:

```
max_sum = 0
for start in range(len(a)):
   for end in range(start, len(a)):
      cur_sum = 0
      for i in range(start, end + 1):
         cur_sum = cur_sum + a[i]
      if cur_sum > max_sum:
         max_sum = cur_sum
```

[1] Bei dem hier verwendeten *Slicing* in Python wird ja die «obere Grenze» als «exklusiv» angenommen. Will man also die Elemente bis *einschliesslich* `a[end]` summieren, muss man im Programm `end + 1` schreiben. Der `range(len(a))` wiederum enthält gerade die benötigten Werte aus dem Intervall *[0…len(a)-1]*.

Schliesslich können auch noch die Python-spezifischen `for`-Schleifen mit den `range`-Funktionen durch grundlegende Schleifenkonstruktionen ersetzt werden, um das hinter dem Programm stehende Verfahren völlig offenzulegen:

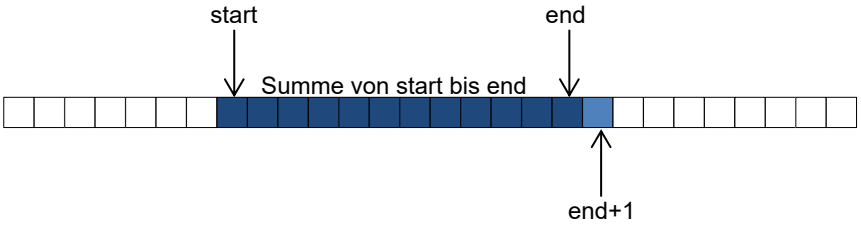
```
max_sum, start = 0, 0
while start < len(a):
   end = start
   while end < len(a):
      cur_sum, i = 0, start
      while i <= end:
         cur_sum, i = cur_sum + a[i], i + 1
      if cur_sum > max_sum:
         max_sum = cur_sum
      end = end + 1
   start = start + 1
```

Programm 2

Betrachtet man die beiden inneren Schleifen in der letzten Version von Programm 1 genauer, stellt man fest, dass die Berechnung der Summe vereinfacht werden kann. Am Ende der innersten Schleife erfüllt die Variable `cur_sum` nämlich folgende Bedingung:

$$cur\_sum = \sum_{i=start}^{end} a[i]$$

Um die Summe der Werte von `start` bis `end+1` zu erhalten, muss lediglich zur gerade zuvor in der innersten Schleife berechneten Summe der Werte von `start` bis `end` ein weiterer Wert des Arrays addiert werden: `cur_sum = cur_sum + a[end + 1]`. Wird dann auch der Wert von `end` um 1 erhöht, gilt wieder der ursprüngliche Zusammenhang zwischen `cur_sum` und `end` – aber eben «eine Stelle weiter links».



Die innerste Schleife kann mithin entfallen, und der neue Wert der Variablen `cur_sum` kann aus dem jeweils vorherigen direkt berechnet werden:

```
max_sum, start = 0, 0
while start < len(a):
   cur_sum, end = 0, start
   while end < len(a):
      cur_sum, end = cur_sum + a[end], end + 1
      if cur_sum > max_sum:
         max_sum = cur_sum
   start = start + 1
```

oder alternativ mit `for`-Schleifen notiert:

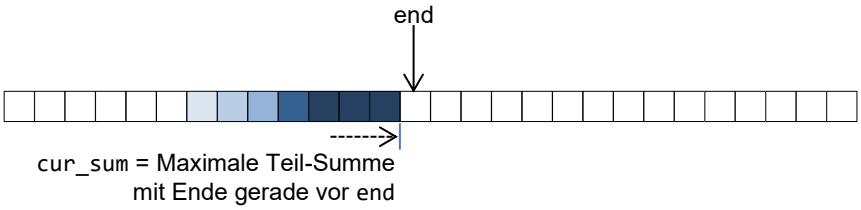
```
max_sum = 0
for start in range(len(a)):
   cur_sum = 0
   for end in range(start, len(a)):
      cur_sum = cur_sum + a[end]
      if cur_sum > max_sum:
         max_sum = cur_sum
```

Programm 3 – Lösung anhand einer Schleifen-Invariante

Ein weiterer Ansatz die Aufgabe zu lösen ergibt sich, wenn man einen anderen Zusammenhang zwischen den beiden Variablen `cur_sum` und `end` annimmt:

> `cur_sum` ist die maximale Summe aller Sequenzen, die *unmittelbar vor* `a[end]` enden.

oder graphisch formuliert:



oder als Formel:

$$cur\_sum = max\left(\bigcup_{start=0}^{end-1} \sum_{i=start}^{end-1} a[i]\right)$$

Wenn also solche Variablen `cur_sum` und `end` wie beschrieben existieren, kann man überlegen, wie man den Wert von `end` um eins vergrössern kann, ohne dass dieser Zusammenhang verloren geht. So kann man dann sukzessive für alle `end`-Werte von 0 bis `len(a)-1` die zugehörigen `cur_sum`-Werte berechnen und prüfen, ob sie grösser sind als das zuvor gefundene Maximum.

Der beschriebene Zusammenhang zwischen den Variablen `cur_sum` und `end` gilt also immer *invariant* immer am Anfang und am Ende jedes Schleifendurchlaufs. Es handelt sich um eine Bedingung, die diese Variablen dann immer erfüllen müssen. Eine solche Bedingung, die unabhängig von der Anzahl der Schleifendurchläufe jedes Mal wieder erfüllt wird, bezeichnet man als *Schleifeninvariante*.

Will man den Wert von `end` um eins erhöhen, muss man für `cur_sum` (zunächst) die maximale Summe mit Sequenz-Ende bei `end` berechnen. Diese ist gerade

$$max(a[end], cur\_sum + a[end])$$

bzw.

$$max(0, cur\_sum) + a[end].$$

Begründung: Das Element `a[end]` muss das letzte Element dieser Sequenz sein. Wenn es eine Sequenz mit Ende an dieser Stelle gibt, deren Wert grösser ist als `a[end]` alleine, dann setzt sich diese Sequenz gerade aus der Sequenz mit maximaler Summe und Ende *unmittelbar vor* `end` und `a[end]` zusammen.

Aus diesen Überlegungen entsteht schliesslich ein Programm in der folgenden Art:

```
max_sum, cur_sum, end = 0, 0, 0
while end < len(a):
   cur_sum, end = (cur_sum + a[end]) if cur_sum > 0 else a[end], end + 1
   if cur_sum > max_sum:
      max_sum = cur_sum
```

oder mit `for`-Schleife formuliert:

```
max_sum, cur_sum = 0, 0
for end in range(len(a)):
   cur_sum = (cur_sum + a[end]) if cur_sum > 0 else a[end]
   if cur_sum > max_sum:
      max_sum = cur_sum
```

Programm 4 – Divide and Conquer-Lösung

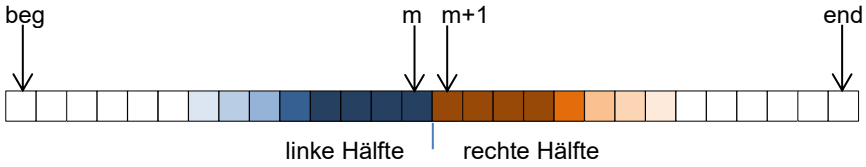
Als letzte Variante soll hier noch eine gezeigt werden, die einem anderen methodischen Ansatz entspringt: «Teile und Beherrsche» oder «Divide and Conquer». Die Grundidee ist, die Sequenz in der Mitte zu teilen. Dann gibt es drei Möglichkeiten, wo sich die gesuchte maximale Sub Sequence befinden kann:

1. vollständig in der vorderen bzw. linken Hälfte
2. vollständig in der hinteren bzw. rechten Hälfte
3. über die Teilungsgrenze hinweg

Die maximalen Sub Sequenzen der beiden Hälften kann man wieder nach demselben Verfahren bestimmen. Die ursprüngliche Aufgabe wird also in zwei kleinere gleichartige und eine dritte einfache Aufgabe geteilt. Da die beiden gleichartigen Aufgaben jetzt aber nur noch halb so lange Sequenzen beinhalten, können die Aufgaben mit geringerem Aufwand gelöst werden.

Kennt man die jeweiligen maximal möglichen Summen aus den beiden Hälften, sowie die maximal mögliche Summe der Sequenzen, welche die Grenze der Teilung beinhalten, kann man davon einfach das Maximum wählen.

Um die gleichartigen «Teile und Beherrsche»-Aufgaben zu lösen, verwendet man eine Funktion, die sich – rekursiv – selbst aufruft. Als Parameter müssen dann die Grenzen der jeweils zu bearbeitenden Sequenz mitgegeben werden. Die wollen wir hier `beg` und `end` nennen. Mit `(beg + end) // 2` kann man dann einen Teilungs-Index in der Mitte berechnen.



Für die dritte Aufgabe sucht und addiert man die maximalen Summen ab Index `m` nach links bis einschliesslich `beg` und nach rechts ab Index `m+1` bis einschliesslich `end`.

```
def max_sub_seq_divide_and_conquer(data):
   def max_in_range(r):
      max_sum, s = 0, 0
      for i in r:
         s = s + data[i]
         if s > max_sum:
            max_sum = s
      return max_sum

   def max_sub_seq_part(beg, end):
      if beg == end:
         return max(0, data[beg])
      else:
         m = (beg + end) // 2
         return max(max_sub_seq_part(beg, m), max_sub_seq_part(m + 1, end),
            max_in_range(range(m, beg - 1, -1)) + max_in_range(range(m + 1, end + 1)))

   return max_sub_seq_part(0, len(data) - 1) if len(data) > 0 else 0
```

Laufzeit-Betrachtungen

Wie schnell laufen die verschiedenen Programme ab? Wie vergleichen Sie sich zueinander?

Die Unterschiede im Laufzeitverhalten sind vor allem für grosse Datenmengen interessant. Für nur wenige Elemente wie z.B. 100 unterscheiden sich die Laufzeiten nicht spürbar. Anders verhält es sich, wenn man immer grössere Datenmengen betrachtet, z.B. indem man die Länge der Eingabe-Liste immer wieder zu verdoppelt. Dann merkt man, dass die Programme sehr verschieden schnell langsamer werden.

Die verschiedenen Verfahren lassen sich dann bald einmal weniger anhand der absoluten Laufzeiten als durch das Wachstum dieser Zeiten klassifizieren. Hier lohnt es, ein wenig zu experimentieren.

Eine formale Analyse-Technik bietet die *asymptotische Komplexität*.