LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND STATISTIK
INSTITUT FÜR INFORMATIK

Skriptum

zur Vorlesung

Algorithmen auf Sequenzen

gehalten im Wintersemester 2020/21

am Lehrstuhl für Bioinformatik

Volker Heun

29. November 2020

Version 8.10

Vorwort

Dieses Skript entstand parallel zur Vorlesung *Algorithmen auf Sequenzen* im Wintersemester 20/21 und baut auf dem vorherigen Skripten der Vorlesungen des Wintersemesters 03/04, des Wintersemesters 04/05, des Wintersemesters 06/07, des Wintersemesters 07/08, des Wintersemesters 09/10, des Sommersemesters 13, des Wintersemesters 15/16 , Wintersemesters 17/18 und desWintersemesters 18/19 auf. Diese Vorlesung wurde an der Ludwig-Maximilians-Universität speziell für Studenten der Bioinformatik, aber auch für Studenten der Informatik, im Rahmen des gemeinsam von der Ludwig-Maximilians-Universität München und der Technischen Universität München veranstalteten Studiengangs Bioinformatik gehalten.

Das vorliegende Skript gibt den Inhalt aller Vorlesungen wieder, die sich jedoch inhaltlich ein wenig unterscheiden. Die Teile, die im Wintersemester 2020/21 nur teilweise behandelt wurden, sind mit einem + markiert und die Teile, die nicht Teil der Vorlesung waren, sind mit einem * markiert.

Diese Fassung ist weitestgehend überarbeitet worden, allerdings kann das Skript immer noch einige (Tipp)Fehler enthalten. Daher bin ich für jeden Hinweis darauf (an Volker.Heun@bio.ifi.lmu.de) dankbar.

An dieser Stelle möchte ich Sabine Spreer, die an der Erstellung des ersten Kapitels in LaTeX 2_ε maßgeblich beteiligt war, sowie Alois Huber und Hermann Klann, die an der Erstellung des zweiten mit sechsten Kapitels in LaTeX 2_ε maßgeblich beteiligt waren, danken. Außerdem bin ich folgenden Personen für Hinweise auf Tippfehler und Verbesserungsmöglichkeiten dankbar: Herrn Martin Bickeböller, Herrn Samuel Klein, Herrn Nick Lehner, Herrn Sebastian Strempel, Herrn Vladimir Viro, Herrn Stefan Weber und Herrn Jeremias Weihmann.

Weiterhin möchte ich insbesondere meinen Mitarbeitern Johannes Fischer, Simon W. Ginzinger, Benjamin Albrecht sowie Caroline Friedel und Marie-Sophie Friedl für Ihre Unterstützung bei den Veranstaltungen danken, die somit das vorliegende Skript erst möglich gemacht haben.

München, im Wintersemester 2020/21 Volker Heun

Inhaltsverzeichnis

1 Optimal Scoring Subsequences **1**

1.1 Maximal Scoring Subsequence 1

1.1.1 Problemstellung 1

1.1.2 Biologische Anwendungen 2

1.1.3 Naive Lösung 4

1.1.4 Lösen durch dynamische Programmierung 4

1.1.5 Divide-and-Conquer-Ansatz 5

1.1.6 Clevere Lösung 6

1.1.7 Zusammenfassung 7

1.2 All Maximal Scoring Subsequences 9

1.2.1 Problemstellung 9

1.2.2 Elementare Eigenschaften der strukturellen Definition 12

1.2.3 Ein Algorithmus zur Lösung 20

1.2.4 Zeitkomplexität 23

1.3 Bounded All Maximum Scoring Subsequences (*) 26

1.3.1 Problemstellung 26

1.3.2 Lösung mittels Dynamischer Programmierung 27

1.3.3 Effiziente Lösung mittels Dynamischer Programmierung 28

1.4 Bounded Maximal Scoring Subsequence (*) 29

1.4.1 Problemstellung 29

1.4.2 Links-Negativität 30

1.4.3 Algorithmus zur Lösung des BMSS-Problems 31

1.5 Maximal Average Scoring Subsequence (*) 34

1.5.1 Problemstellung . . . 34
1.5.2 Rechtsschiefe Folgen und fallend rechtsschiefe Partitionen . . . 34
1.5.3 Algorithmus zur Konstruktion rechtsschiefer Zeiger . . . 38
1.5.4 Elementare Eigenschaften von MASS . . . 40
1.5.5 Ein Algorithmus für MASS . . . 41
1.6 Weighted Maximal Average Scoring Subsequence (*) . . . 45
1.6.1 Problemstellung . . . 45
1.6.2 Elementare Eigenschaften . . . 46
1.6.3 Generischer Algorithmus und seine Korrektheit . . . 47
1.6.4 Linksschiefe Folgen und steigend linksschiefe Partitionen . . . 51

2 Suffix Trees Revisited **57**
2.1 Definition von Suffix Tries und Suffix Trees . . . 57
2.1.1 Σ-Bäume und (kompakte) Σ^+-Bäume . . . 57
2.1.2 Grundlegende Notationen und elementare Eigenschaften . . . 58
2.1.3 Suffix Tries und Suffix Trees . . . 59
2.2 Repräsentationen von Bäumen . . . 62
2.2.1 Darstellung der Kinder mit Feldern . . . 62
2.2.2 Darstellung der Kinder mit Listen . . . 63
2.2.3 Darstellung der Kinder mit balancierten Bäumen . . . 64
2.2.4 Darstellung des Baumes mit einer Hash-Tabelle . . . 65
2.2.5 Speicherplatzeffiziente Feld-Darstellung . . . 66
2.3 WOTD-Algorithmus . . . 67
2.3.1 Die Konstruktion . . . 67
2.3.2 Zeitbedarf . . . 69
2.4 Der Algorithmus von Ukkonen . . . 70
2.4.1 Suffix-Links . . . 70

2.4.2 Verschachtelte Suffixe und verzweigende Teilwörter 72
2.4.3 Idee von Ukkonens Algorithmus 73
2.4.4 Ukkonens Online Algorithmus 76
2.4.5 Zeitanalyse 82

3 Repeats **85**
3.1 Exakte und maximale Repeats 85
3.1.1 Erkennung exakter Repeats 85
3.1.2 Charakterisierung maximaler Repeats 88
3.1.3 Erkennung maximaler Repeats 92
3.1.4 Revers-komplementäre Repeats 94
3.2 Tandem-Repeats und Suffix-Bäume 96
3.2.1 Was sind Tandem-Repeats 96
3.2.2 Eigenschaften von Tandem-Repeats 98
3.2.3 Algorithmus von Stoye und Gusfield 100

A Literaturhinweise **103**
A.1 Lehrbücher zur Vorlesung 103
A.2 Skripten anderer Universitäten 103
A.3 Originalarbeiten 104
A.3.1 Optimal Scoring Subsequences 104
A.3.2 Suffix-Trees 105
A.3.3 Repeats 105
A.3.4 Lowest Common Ancestors and Range Minimum Queries . . . 106
A.3.5 Construction of Suffix-Arrays 107
A.3.6 Applications of Suffix-Arrays 108
A.3.7 Sorting by Reversals 109
A.3.8 Sorting by Oriented Reversals 110

A.3.9 Sorting by Transpositions 111
A.3.10 Sorting by Transversals 111
A.3.11 Erweiterungen zu Genome Rearrangements 112

B Index **115**

Optimal Scoring Subsequences 1

1.1 Maximal Scoring Subsequence

Ziel dieses Abschnittes ist es, (möglichst effiziente) Algorithmen für das Maximal Scoring Subsequence Problem vorzustellen. Dabei werden wir zunächst noch einmal kurz die wichtigsten Paradigmen zum Entwurf von Algorithmen wiederholen.

1.1.1 Problemstellung

Maximal Scoring Subsequence (MSS)

Eingabe: Eine Folge $(a_1, \ldots, a_n) \in \mathbb{R}^n$.
Gesucht: Eine (zusammenhängende) Teilfolge $(a_i, \ldots, a_j)$, die $\sigma(i, j)$ maximiert, wobei $\sigma(i, j) = \sum_{\ell=i}^{j} a_\ell$.

Bemerkung: Mit Teilfolgen sind in diesem Kapitel immer (sofern nicht anders erwähnt) zusammenhängende (d.h. konsekutive) Teilfolgen einer Folge gemeint (also anders als beispielsweise in der Analysis).

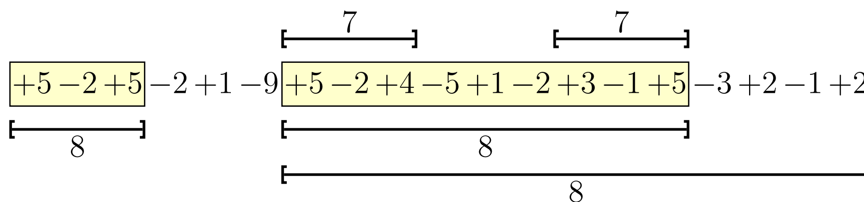


Abbildung 1.1: Beispiel: Maximal Scoring Subsequences

In Abbildung 1.1 ist ein Beispiel angegeben. Wie man dort sieht, kann es mehrere (und auch nicht-disjunkte) Lösungen geben.

Bemerkungen:

- Es sind mehrere Lösungen möglich.
- Die leere Folge mit Score 0 interpretieren wir auch als eine Lösung.

- Ist eine Lösung in der anderen enthalten, so wählen wir als Lösung immer eine kürzester Länge. Die anderen ergeben sich aus Anhängen von Teilfolgen mit dem Score Null. Darüber hinaus haben solche Lösungen noch eine schöne Eigenschaft, wie wir gleich sehen werden.

- Es gibt keine echt überlappenden Lösungen. Angenommen, es gäbe echt überlappende Lösungen a' und a'' einer gegebenen Folge a (siehe dazu auch Abbildung 1.2). Die Sequenz gebildet aus der Vereinigung beider Sequenzen (respektive ihrer Indices) müsste dann einen höheren Score haben, da der Score der Endstücke > 0 ist (sonst würde er in den betrachteten Teilfolgen nicht berücksichtigt werden).

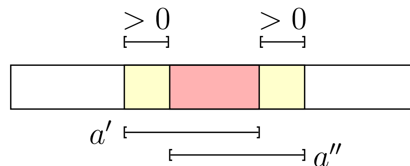


Abbildung 1.2: Skizze: Überlappende optimale Teilfolgen

1.1.2 Biologische Anwendungen

In diesem Abschnitt wollen wir kurz einige biologische Probleme vorstellen, die sich als Maximal Scoring Subsequence formulieren lassen.

Transmembranproteine: Bestimmung der transmembranen Regionen eines Proteins. Eingelagerte Proteine in der Membran sollten einen ähnlichen Aufbau wie die Membran selbst haben, da die Gesamtstruktur stabiler ist. Somit sollten transmembrane Regionen hydrophob sein.

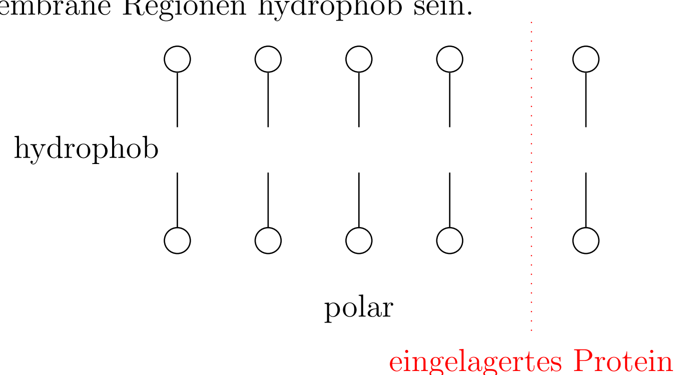


Abbildung 1.3: Beispiel: transmembrane Proteine

Mit einer geeigneten Gewichtung der Aminosäuren, können solche hydrophoben Regionen mit Hilfe der Lösung eines Maximal Scoring Subsequence Problems gefunden werden.

Für die einzelnen Aminosäuren werden die folgende Werte gemäß der Hydrophobizität der entsprechenden Aminosäure gewählt:

- hydrophobe Aminosäuren: ein Wert aus $\in [0 : 3]$;
- hydrophile Aminosäuren: ein Wert aus $\in [-5 : 0]$.

Lokalisierung GC-reicher DNS-Abschnitte: In GC-reichen Regionen der DNS finden sich häufig Gene. Das Auffinden solcher GC-reicher Regionen lässt sich als Maximal Scoring Subsequence Problem beschreiben:

- $C, G \to 1 - p$ für ein $p \in [0 : 1]$;
- $A, T \to -p$.

Zusätzlich können Längenbeschränkungen sinnvoll sein; obere, untere Schranke der Länge für z.B. Proteine, die man sucht.

Vergleichende Analyse von Genomen: Im Vergleich des Mensch- und Maus-Genoms liegen Sequenz-Ähnlichkeiten für Exons bei 85% und für Introns bei 35%. Mit Hilfe eines lokalen Sequenz-Alignments (Smith-Waterman) lassen sich solche Übereinstimmungen gut auffinden. Jedoch kann es bei den gefundenen Lösungen den so genannten Mosaik-Effekt geben, d.h. sehr ähnliche Sequenzen sind immer wieder von sehr unähnlichen, jedoch relativ kurzen Stücken unterbrochen.

Mit Hilfe eines geeigneten Maximal Scoring Subsequences Problems können solche Mosaik-Effekte aufgedeckt werden. Hierzu wird eine Variante des Maximal Scoring Subsequence Problems verwendet. Man normiert die erzielten Scores mit der Länge der zugehörigen Teilfolge. Somit lassen sich so genannte *poor regions* (stark positive Teile, die mit kurzen stark negativen Fragmenten unterbrochen sind) ausschließen.

Konservierte Regionen: Gut konservierte Regionen eines mehrfachen Sequenzenalignments lassen sich durch Gewichtung der Spalten gemäß ihrer Ähnlichkeiten (beispielsweise SP-Maß einer Spalte) und einem anschließenden Auffinden von Maximal Scoring Subsequences bestimmen.

'Ungapped' local alignment: Auch lokales Alignment ohne Lücken (gaps) können aus der Dot-Matrix durch Anwenden von Algorithmen für das Maximal Scoring Subsequence Problem auf die Diagonalen effizient finden. Dieses Verfahren ist insbesondere dann effizient, wenn man mit Längenrestriktionen arbeiten will, oder den Score ebenfalls wieder mit der zugehörigen Länge der Folge normalisieren will.

1.1.3 Naive Lösung

Im Folgenden wollen wir eine Reihe von Algorithmen zur Lösung des Maximal Scoring Subsequence Problems vorstellen, die jeweils effizienter als die vorher vorgestellte Variante ist.

Die naive Methode bestimmt zuerst alle Werte $\sigma(i,j)$ für alle $i \leq j \in [1:n]$. Anschließend wird aus den Werten ein Maximum ermittelt, ggf. eines dessen zugehörige Sequenz die kürzeste Länge aufweist.

Für die Laufzeit (Anzahl Additionen, die proportional zur Laufzeit ist) ergibt sich dann pro Tabelleneintrag $\Theta(j-i)$, also insgesamt:

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\Theta(j-i) = \Theta\left(\sum_{i=1}^{n}\sum_{j=0}^{n-i} j\right) = \Theta\left(\sum_{i=1}^{n}(n-i)^2\right) = \Theta\left(\sum_{i=1}^{n} i^2\right) = \Theta(n^3).$$

Ein alternative Begründung ist die folgende: In der Tabelle mit n^2 Einträgen benötigt jeder Eintrag maximal n Operationen. Hierbei erhalten wir jedoch nur eine obere Schranke und nicht die korrespondierende untere Schranke für die Laufzeit.

1.1.4 Lösen durch dynamische Programmierung

Ein anderer Ansatz ergibt sich aus einer trivialen Rekursionsgleichung für $\sigma(i,j)$. Es gilt folgende Rekursionsgleichung:

$$\sigma(i,j) = \begin{cases} a_i & \text{für } i = j \\ \sigma(i,k) + \sigma(k+1,j) & \text{für ein } k \in [i:j-1] \text{ sofern } i < j \end{cases}$$

In der Regel ist eine direkte Implementierung dieser Rekursionsgleichung zu aufwendig, da meist exponentiell viele Aufrufe erfolgen (siehe zum Beispiel rekursive Berechnung einer Fibonacci-Zahl)! In diesem Fall wären es sogar nur n^3 rekursive Aufraufe, was aber nicht besser als der naive Ansatz ist. Mit Hilfe der *dynamische Programmierung* können wir jedoch effizienter werden, da hier Werte mehrfach berechnet werden.

Die Tabellengröße ist $O(n^2)$. Jeder Eintrag kann mit der Rekursionsgleichung in Zeit $O(1)$ berechnet werden. Dabei wird die Tabelle beginnend von der Mitteldiagonalen aus über alle Nebendiagonalen zur rechten oberen Ecke hin aufgefüllt (siehe auch Abbildung 1.4).

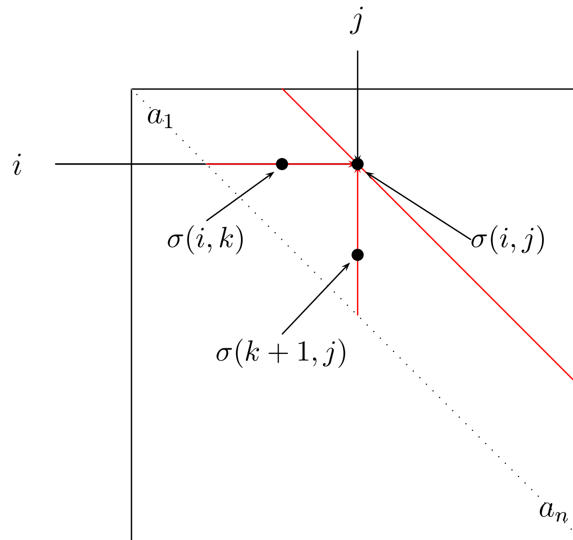


Abbildung 1.4: Skizze: Auffüllen der dynamischen Programmierungstabelle

1.1.5 Divide-and-Conquer-Ansatz

Eine andere Lösungsmöglichkeit erhalten wir einem Divide-and-Conquer-Ansatz, wie in Abbildung 1.5 illustriert. Dabei wird die Folge in zwei etwa gleich lange Folgen aufgeteilt und die Lösung in diesen rekursiv ermittelt.

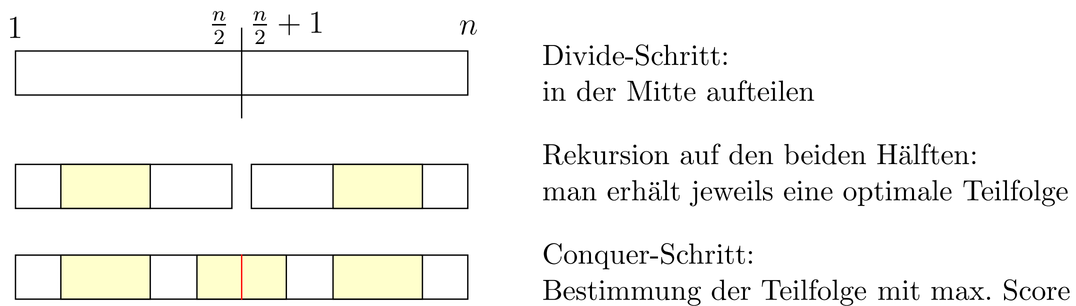
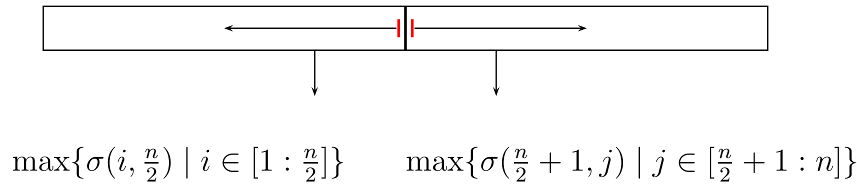


Abbildung 1.5: Skizze: Divide-and-Conquer bei Maximal Scoring Subsequences

Man kann dabei aber auch die optimale Teilfolge in der Mitte zerschneiden, die dann die Elemente $a_{n/2}$ und $a_{n/2+1}$ enthalten muss. Daher muss man zusätzlich von der Mitte aus testen, wie von dort nach rechts bzw. links eine optimale Teilfolge aussieht (siehe auch Abbildung 1.6).

Dazu bestimmen wir jeweils das Optimum der Hälften, d.h

$$\max\left\{\sigma\left(i,n/2\right) \mid i \in [1:n/2]\right\} \qquad \text{und} \qquad \max\left\{\sigma\left(n/2+1,j\right) \mid j \in [n/2+1:n]\right\}.$$



$$\max\{\sigma(i, \tfrac{n}{2}) \mid i \in [1 : \tfrac{n}{2}]\} \qquad \max\{\sigma(\tfrac{n}{2}+1, j) \mid j \in [\tfrac{n}{2}+1 : n]\}$$

Abbildung 1.6: Skizze: Conquer-Step

Man überlegt sich leicht, dass die optimale Teilfolge, die die Positionen $n/2$ und $n/2 + 1$ überdeckt, aus der Konkatenation der beiden berechneten optimalen Teilfolgen in den jeweiligen Hälften bestehen muss.

Für die Laufzeit erhalten wir sofort die folgende Rekursionsgleichung, die identisch zur Laufzeitanalyse von Mergesort ist, da das Bestimmen einer optimalen Teilfolge über die Mitte hinweg in Zeit $O(n)$ (wie oben gesehen) geschehen kann:

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

1.1.6 Clevere Lösung

Wenn wir wie beim Divide-and-Conquer-Ansatz das Problem nicht in der Mitte aufteilen, sondern am rechten Rand, so können wir (ganz im Gegensatz zum Problem des Sortierens) eine effizientere Lösung finden (siehe auch Abbildung 1.7).

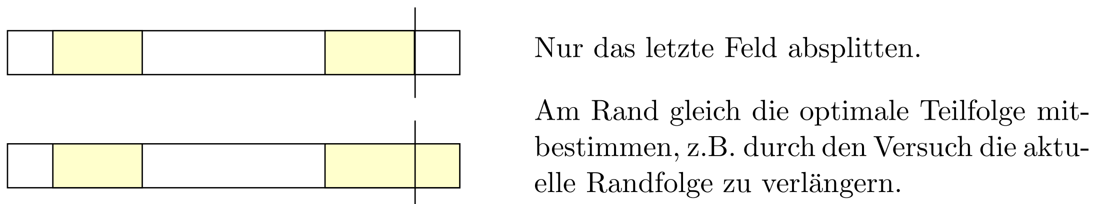


Abbildung 1.7: Skizze: Asymmetrische Divide-and-Conquer

Damit wir nun beim Conquer-Schritt die zusätzliche Zeit einsparen können, bestimmen wir in der linken Rekursion noch die optimale Teilfolge mit, die die letzte Position beinhaltet. Im Conquer-Step wird dann diese Lösung am letzten Rand (die die Position $n-1$ beinhaltet) um die Position n erweitert, wenn der Wert der rekursiven Lösung positiv ist. Andernfalls ist die Folge a_n die gesuchte Lösung, die die Position n beinhaltet.

Man kann diese Idee rekursiv als Divide-and-Conquer implementieren oder iterativ, wie in Abbildung 1.8 angegeben, auflösen. Da im Wesentlichen einmal linear über die Folge gelaufen wird und für jedes Element nur konstante Kosten (Additionen,

```
MSS (real a[], int n)
begin
    int max := 0, ℓ := 1, r := 0 ;                  /* best global solution */
    int rmax := 0, rstart := 1 ;                 /* best solution at right end */
    for (i := 1; i ≤ n; i++) do
        if (rmax > 0) then                       /* equivalent to rmax + a_i > a_i */
            rmax := rmax + a_i;
        else
            rmax := a_i;
            rstart := i;
        if (rmax > max) then
            max := rmax;
            ℓ := rstart;
            r := i;
    return (ℓ, r, max);
end
```

Abbildung 1.8: Algorithmus: Die clevere Lösung

Maximumsbildungen) anfallen, erhalten wir offensichtlich eine Laufzeit von $O(n)$. Da dies eine offensichtlich optimale Lösung ist, halten wir das Ergebnis im folgenden Satz fest.

Theorem 1.1 *Eine Teilfolge mit maximalem Wert einer gegebenen reellen Folge lässt sich in Linearzeit mit konstantem zusätzlichen Platzbedarf bestimmen.*

1.1.7 Zusammenfassung

In der folgenden Tabelle in Abbildung 1.9 sind alle Resultate der vorgestellten Algorithmen noch einmal zusammengefasst.

Algorithmus	Zeit	Platz	Bemerkung
Naiver Algorithmus	$O(n^3)$	$O(n^2)$	Tabelle füllen
Dyn. Programmierung	$O(n^2)$	$O(n^2)$	geschickter füllen
Divide and Conquer	$O(n \log(n))$	$O(n)$	die Eingabelänge ist n
Clevere Lösung	$O(n)$	$n + O(1)$	die Eingabelänge ist n

Abbildung 1.9: Tabelle: Laufzeiten für die MSS Lösungsansätze

In der folgenden Tabelle in Abbildung 1.10 sind explizit die Längen von Folgen angegeben, die in einer Sekunde bzw. einer Minute auf einem gewöhnlichen Rechner mit Stand 2004 (AMD-Athlon, 2GHz, 2GB Hauptspeicher) verarbeitet werden können (mittels eines C-Programms).

Seq-Len	1sec.		1min.
Naive	800	$\xrightarrow{\times 4}$	3.200
Dyn.Prog.	4.200	$\xrightarrow{\times 8}$	32.000
D&C	1.500.000	$\xrightarrow{\times 50}$	75.000.000
Clever	20.000.000	$\xrightarrow{\times 60}$	*1.200.000.000*

Abbildung 1.10: Tabelle: zu verarbeitenden Problemgrößen (Stand 2004)

In der folgenden Tabelle in Abbildung 1.11 ist das gleiche Experiment mit Stand 2008 (Intel Dual-Core, 2.4GHz, 4GB Hauptspeicher) angegeben.

Seq-Len	1sec.		1min.
Naive	1.325	$\xrightarrow{\times 4}$	5.200
Dyn.Prog.	27.500	$\xrightarrow{\times 8}$	220.000
D&C	10.000.000	$\xrightarrow{\times 50}$	500.000.000
Clever	400.000.000	$\xrightarrow{\times 60}$	*24.000.000.000*

Abbildung 1.11: Tabelle: zu verarbeitenden Problemgrößen (Stand(2008)

In der Tabelle in der Abbildung 1.12 ist da gleiche Experiment mit Stand 2014 (Intel Quad Core i7-3770, 3.40 GHz, 32 GB Hauptspeicher) angegeben.

Seq-Len	1sec.		1min.
Naiv	1.500	$\xrightarrow{\times \approx 4}$	5.800
Dyn.Prog.	36.000	$\xrightarrow{\times \approx 8}$	280.000
D&C	18.000.000	$\xrightarrow{\times \approx 50}$	950.000.000
Clever	650.000.000	$\xrightarrow{\times \approx 60}$	*40.000.000.000*

Abbildung 1.12: Tabelle: zu verarbeitenden Problemgrößen (Stand 2014)

In der Tabelle in der Abbildung 1.13 ist da gleiche Experiment mit Stand 2019 (Intel(R) Core(TM) i7-9700,3.00 GHz, 642 GB Hauptspeicher) angegeben.

Seq-Len	1sec.		1min.
Naiv	2.100	$\xrightarrow{\times \approx 4}$	8.300
Dyn.Prog.	65.000	$\xrightarrow{\times \approx 8}$	500.000
D&C	36.000.000	$\xrightarrow{\times \approx 50}$	2.000.000.000
Clever	420.000.000	$\xrightarrow{\times \approx 60}$	*25.000.000.000*

Abbildung 1.13: Tabelle: zu verarbeitenden Problemgrößen (Stand 2019)

Man sieht, dass sich bei den einfachen Algorithmen (Naiv und Dynamische Programmierung) die Größenordnungen kaum ändern, bei den geschickten Varianten hingegen schon. Weiterhin stellt man fest, dass beim linearen Ansatz die Leistung von 2019 gegenüber 2014 zurückgegangen ist. Dies liegt vermutlich in den neuen Architekturen und dem niedrigeren Takt begründet, mit denen einfache Operationen etwas langsamer werden, während komplexe Operationen weiterhin besser vorberechnet werden.

Beachte hierbei, dass die Faktoren bei einer Versechzigfachung der Laufzeit beim naive Algorithmus etwa $\sqrt[3]{60} \approx 4$, bei der dynamischen Programmierung etwa $\sqrt{60} \approx 8$ und beim Cleveren Algorithmus etwa 60 ist. Die Funktion $n \log(n)$ besitzt keine einfache anzugebende Inverse, der Wert 50 entspricht aber in etwa der Inversen für 60 im Bereich der betrachteten Werte von n.

Die kursiven Werte in den Tabellen sind geschätzt, da der benötigte Hauptspeicher nicht zur Verfügung stand.

1.2 All Maximal Scoring Subsequences

Nun wollen wir uns mit der Frage beschäftigen, wenn wir nicht nur eine beste, sondern alle besten bzw. alle möglichen Teilfolgen mit positive Score und zwar nach absteigenden Score erhalten wollen. Zuerst einmal müssen wir uns über die Fragestellung klar werden, d.h. was ist überhaupt die gesuchte Lösung.

1.2.1 Problemstellung

Geben wir zunächst die formale Problemstellung an und diskutieren anschließend die damit verbundenen Probleme.

All Maximal Scoring Subsequences (AMSS)

Eingabe: Eine Folge $(a_1, \ldots, a_n) \in \mathbb{R}^n$.
Gesucht: Alle disjunkten Teilfolgen von a, die ihren Score maximieren.

Zunächst einmal muss man sich überlegen, was es heißen soll, dass *alle disjunkten Teilfolgen ihren Score maximieren.* Betrachten wir dazu das Beispiel in Abbildung 1.14. Die lange Folge ist keine Lösung, da wir keine überlappenden Folgen haben wollen.

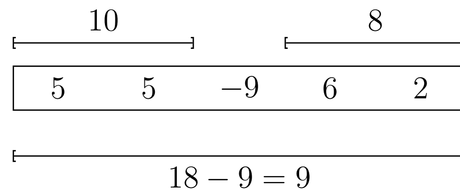


Abbildung 1.14: Beispiel: Maximal bewertete Teilfolgen

Wir geben zwei mögliche Definitionen an, wie man alle maximalen Teilfolgen einer Folge definieren kann. Im Folgenden werden wir zeigen, dass die beiden Definitionen äquivalent sind. A priori ist dies überhaupt nicht klar.

Definition 1.2 (Strukturelle Definition) *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge. Eine Teilfolge $a' = (a_i, \ldots, a_j)$ von a heißt* maximal bewertet *(engl. maximal scoring), wenn die beiden folgenden Eigenschaften erfüllt sind:*

(E1) alle echten Teilfolgen von a' haben einen kleineren Score;

(E2) keine echte Oberfolge von a' in a erfüllt E1.

Die Bedingung E1 ist nach unserer vorhergehenden Diskussion klar, da wir keine Teilfolge mit größerem Score, die Teilfolge einer anderen Teilfolge ist, als Lösung verlieren wollen.

Als Bedingung E2 würde man vermutlich zunächst erwarten, dass jede Oberfolge ebenfalls einen kleineren Score als die gegebene Teilfolge besitzen soll. Die ist jedoch zu naiv, wie das vorherige Beispiel mit $a = (5, 5, -9, 6, 2)$ zeigt. Damit wäre die Teilfolge $a' = (6, 2)$ keine maximal bewertete Teilfolge, da die Oberfolge $a = (a_1, \ldots, a_5)$ einen höheren Score besitzt: $9 > 8$. Als Lösungsmenge würde man jedoch sicherlich $\text{MSS}(a) = \{(a_1, a_2), (a_4, a_5)\}$ erwarten. Diese beiden Folgen erfüllen jedoch sowohl E1 als auch E2.

Halten wir jetzt die endgültige Definition noch fest.

All Maximal Scoring Subsequences (AMSS)

Eingabe: Eine Folge $(a_1, \ldots, a_n) \in \mathbb{R}^n$.
Gesucht: Alle maximal bewerteten Teilfolgen von a.

Neben dieser strukturellen Definition kann man auch noch eine eher algorithmisch angelehnte Definition angeben.

Definition 1.3 (Prozedurale Definition) *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge. Eine kürzeste Teilfolge $(a_i, \ldots, a_j)$ von a mit maximalem Score heißt* maximal bewertet *(engl. maximal scoring). Teilfolgen aus $(a_1, \ldots, a_{i-1})$ bzw. $(a_{j+1}, \ldots, a_n)$, die für diese maximal bewertet sind, sind auch für a* maximal bewertet.

Man überlegt sich leicht, dass die Menge aller maximal bewerteten Teilfolgen nach der prozeduralen Definition eindeutig ist. Wir werden später noch sehen, dass dies auch für die strukturelle Definition gilt (das folgt aus der noch zu zeigenden Äquivalenz der beiden Definitionen).

Notation 1.4 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge, dann bezeichnet $MSS(a)$ die Menge aller maximal bewerteten Teilfolgen von a.*

Aus der prozeduralen Definition kann sofort ein rekursiver Algorithmus zur Bestimmung aller maximal bewerteter Teilfolgen abgeleitet werden, der in der folgenden Skizze in Abbildung 1.15 veranschaulicht ist. Man bestimmt zunächst eine Maximal Scoring Subsequence und entfernt diese aus der Folge. Für die beiden entstehenden Folgen wird dieser Algorithmus rekursiv aufgerufen.

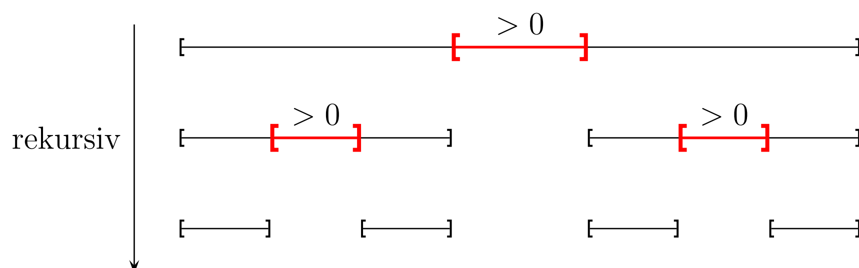


Abbildung 1.15: Skizze: Rekursiver Ansatz für AMSS

Die Laufzeit dieses Algorithmus erfüllt folgende Rekursionsgleichung, da das Auffinden einer Maximal Scoring Subsequence, wie wir im letzten Abschnitt gesehen haben, in Zeit $O(n)$ durchführbar ist:

$$T(n) = O(n) + T(n_1) + T(n_2) \quad \text{mit} \quad n_1 + n_2 < n.$$

Ähnlich zu Quicksort ergibt sich folgende Analyse. Im worst-case benötigt dieser Algorithmus offensichtlich maximal Zeit $O(n^2)$.

Im average-case kann man aus der Analyse von Quicksort herleiten, dass auch dieser Algorithmus einen Zeitbedarf von $O(n \log(n))$ hat. Hierzu muss man jedoch eine geeignete Wahrscheinlichkeitsverteilung annehmen, die in der Rechnung äquivalent zu der von Quicksort ist, die aber nicht unbedingt realistisch sein muss!

1.2.2 Elementare Eigenschaften der strukturellen Definition

In diesem Abschnitt werden wir einige grundlegende Eigenschaften maximal bewerteter Teilfolgen nach der strukturellen Definition herleiten, die es uns erlauben werden, die Äquivalenz der beiden Definitionen maximal bewerteter Teilfolgen zu zeigen. Darauf basierend werden wir im nächsten Abschnitt einen effizienten Algorithmus für die Lösung des All Maximal Scoring Subsequences Problem vorstellen.

Lemma 1.5 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Für jede Teilfolge $a' = (a_i, \ldots, a_j)$ von a sind äquivalent:*

1) a' erfüllt E1.

2) Es gilt für das Minimum aller Präfixe von a in a', dass

$$\sigma(1, i-1) = \min\{\sigma(1, k) \mid k \in [i-1, j]\},$$

und für das Maximum aller Präfixe von a in a', dass

$$\sigma(1, j) = \max\{\sigma(1, k) \mid k \in [i-1, j]\},$$

und dass diese jeweils eindeutig sind!

3) $\forall k \in [i : j] : \sigma(i, k) > 0 \wedge \sigma(k, j) > 0$.

Beweis: 1 ⇒ 2 : Wir führen den Beweis durch Widerspruch.

<u>Für das Maximum:</u> Sei $k \in [i-1 : j-1]$ mit $\sigma(1, k) \geq \sigma(1, j)$. Dann ergibt sich die folgende Situation, die in Abbildung 1.16 dargestellt ist.

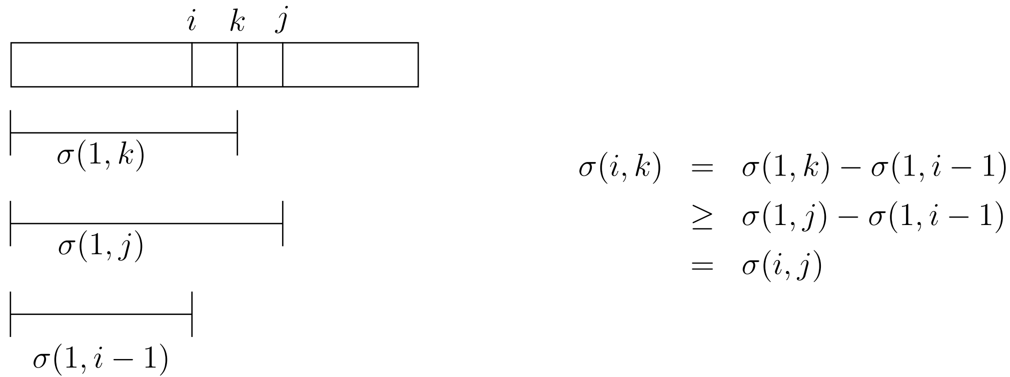


Abbildung 1.16: Skizze: Fall 1: $\sigma(1, k) \geq \sigma(1, j)$

Daraus ergibt sich ein Widerspruch zur Annahme der Eigenschaft E1, da die echte Teilfolge $(a_i, \ldots, a_k)$ von $(a_i, \ldots, a_j)$ keinen kleineren Score besitzt.

Anmerkung: Der Beweis gilt auch für $k = i - 1$. In diesem Fall wäre $\sigma(i, j) \leq 0$, was nicht sein kann.

Für das Minimum: Sei jetzt $k \in [i : j]$ mit $\sigma(1, k) \leq \sigma(1, i - 1)$. Dann ergibt sich die folgende Situation, die in Abbildung 1.17 dargestellt ist.

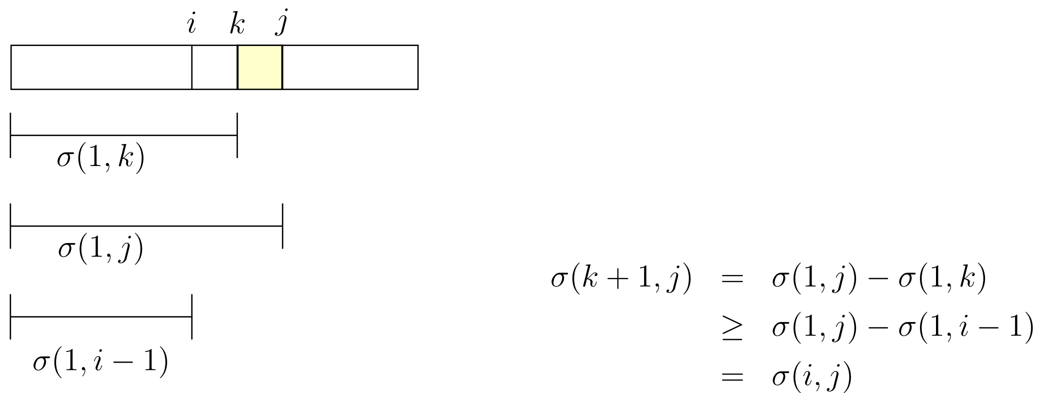


Abbildung 1.17: Skizze: $\sigma(1, k) \leq \sigma(1, i - 1)$

Daraus ergibt sich ein Widerspruch zur Annahme der Eigenschaft E1, da die echte Teilfolge $(a_{k+1}, \ldots, a_j)$ von $(a_i, \ldots, a_j)$ keinen kleineren Score besitzt.

Anmerkung: Der Beweis gilt auch für $k = j$. In diesem Fall wäre $\sigma(i, j) \leq 0$, was nicht sein kann.

2 ⇒ 3 : Da das Minimum eindeutig ist, folgt $\sigma(1, i-1) < \sigma(1, k)$ für alle $k \in [i : j]$ und somit:

$$\sigma(i, k) = \sigma(1, k) - \underbrace{\sigma(1, i-1)}_{<\sigma(1,k)} > 0.$$

Da das Maximum eindeutig ist, folgt $\sigma(1, j) > \sigma(1, k-1)$ für alle $k \in [i : j]$ und somit:

$$\sigma(k, j) = \underbrace{\sigma(1, j)}_{>\sigma(1,k-1)} - \sigma(1, k-1) > 0.$$

3 ⇒ 1 : Sei $a'' = (a_k, \ldots, a_\ell)$ mit $i \leq k \leq \ell \leq j$ sowie $i \neq k$ oder $\ell \neq j$. Dann gilt:

$$\sigma(k, \ell) = \sigma(i, j) - \underbrace{\sigma(i, k-1)}_{\geq 0} - \underbrace{\sigma(\ell + 1, j)}_{\geq 0} < \sigma(i, j).$$

Hinweis: $\sigma(i, k-1)$ und $\sigma(\ell+1, j)$ sind jeweils größer oder gleich 0, eines davon muss jedoch echt größer als 0 sein, da ansonsten $k = i$ und $\ell = j$ gilt und somit $a' = a''$ gelten würde.

Damit ist gezeigt, dass E1 gilt. ■

Lemma 1.6 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Die maximal bewerteten Teilfolgen von a sind paarweise disjunkt.*

Beweis: Wir führen auch hier den Beweis durch Widerspruch. Seien a' und a'' zwei maximal bewertete Teilfolgen von a. Nehmen wir zunächst an, dass a' eine Teilfolge von a'' ist. Dies kann jedoch nicht sein, da dann a'' eine Oberfolge von a' ist, die E1 erfüllt (da a'' eine maximal bewertete Teilfolge ist). Also erfüllt a' nicht E2 und kann somit keine maximal bewertete Teilfolge sein und wir erhalten den gewünschten Widerspruch. Der Fall, dass a'' eine Teilfolge von a' ist, ist analog.

Seien alsoe a' und a'' jetzt zwei maximal bewertete Teilfolgen von a, die sich überlappen. Dies ist in Abbildung 1.18 illustriert. Wir zeigen jetzt, dass die Folge a''', die als Vereinigung der Folgen a' und a'' (respektive ihrer Indizes) definiert ist, die Eigenschaft E1 erfüllt. Sei dazu b eine beliebige Teilfolge von a'''. Ist b eine Teilfolge von a' (bzw. a'') dann gilt aufgrund der Eigenschaft E1 von a' (bzw. a'') $\sigma(b) < \sigma(a') < \sigma(a''')$ (bzw. $\sigma(b) < \sigma(a') < \sigma(a'''))$.

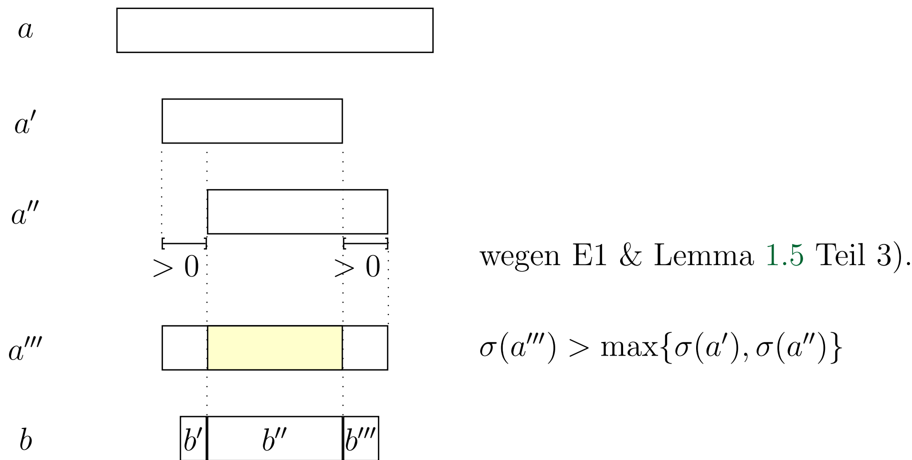


Abbildung 1.18: Skizze: Zwei überlappende maximal bewerteten Teilfolgen

Sei also nun b keine Teilfolge von a' oder a''. Sei weiter $b = b' \cdot b'' \cdot b'''$, so dass $b' \cdot b''$ ein Suffix von a' und $b'' \cdot b'''$ ein Präfix von a'' ist. Dann gilt:

$$\begin{aligned}
\sigma(b) &= \sigma(b' \cdot b'') + \sigma(b'' \cdot b''') - \sigma(b'') \\
&\quad \text{da } \sigma(b' \cdot b'') < \sigma(a') \text{ wegen E1 von } a' \\
&\quad \text{und } \sigma(b'' \cdot b''') < \sigma(a'') \text{ wegen E1 von } a'' \\
&< \sigma(a') + \sigma(a'') - \sigma(b'') \\
&= \sigma(a''')
\end{aligned}$$

Dies ergibt den gewünschten Widerspruch zur Eigenschaft E2 von a' und a'', da a''' E1 erfüllt und eine Oberfolge von a' und a'' ist. ■

05.11.20

Lemma 1.7 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Jede Teilfolge $a' = (a_k, \ldots, a_\ell)$, die E1 erfüllt, ist Teilfolge einer maximal bewerteten Teilfolge.*

Beweis: Wir führen den Beweis durch Widerspruch. Dazu sei $a' = (a_k, \ldots, a_\ell)$ ein *längstes* Gegenbeispiel. Also a' erfüllt E1 mit maximaler Länge, d.h. $\ell - k$ ist unter allen Gegenbeispielen maximal. Dies ist in Abbildung 1.19 illustriert.

Somit erfüllt a' die Eigenschaft E1, aber **nicht** E2 (sonst wäre a' kein Gegenbeispiel). Somit existiert eine echte Oberfolge $a'' = (a_i, \ldots, a_j)$ von a', die E1 erfüllt. Würde die Folge a'' auch E2 erfüllen, dann wäre a'' eine maximal bewertete Teilfolge, die auch eine Oberfolge von a' ist, d.h. a' wäre kein Gegenbeispiel.

Also erfüllt a'' die Eigenschaft E1, aber nicht E2. Besäße a'' eine maximal bewertete Oberfolge, so wäre diese auch eine maximal bewertete Oberfolge von a' und a' somit kein Gegenbeispiel.

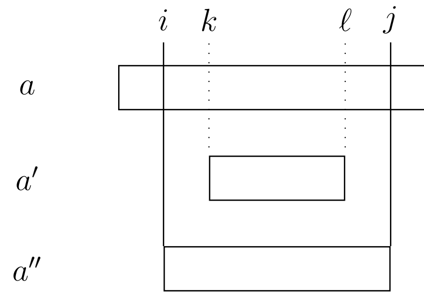


Abbildung 1.19: Skizze: a' ist ein Gegenbeispiel maximaler Länge

Also besitzt a'' keine maximal bewertete Oberfolge und erfüllt E1. Somit ist a'' ein längeres Gegenbeispiel als a', da $j - i > \ell - k$. Dies führt zu einem Widerspruch zur Annahme. ■

Korollar 1.8 *Jedes positive Element ist in einer maximal bewerteten Teilfolge enthalten.*

Beweis: Dies folgt aus dem vorherigen Lemma, da jede einelementige Folge mit einem positiven Wert die Eigenschaft E1 erfüllt. ■

Korollar 1.9 *Innerhalb jeder Teilfolge, die mit keiner maximal bewerteten Teilfolge überlappt, sind die aufaddierten Scores monoton fallend.*

Beweis: Nach dem vorherigen Korollar müssen alle Elemente einer solchen Folge nichtpositiv sein. ■

Lemma 1.10 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Betrachte $\mu = \min\{\sigma(1, k) \mid k \in [0 : n]\}$ (bzw. $\mu = \max\{\sigma(1, k) \mid k \in [0 : n]\}$). Sei k der größte (bzw. kleinste) Wert mit $\sigma(1, k) = \mu$. Dann beginnt an Position $k+1$ (bzw. endet an Position k) eine maximal bewertete Teilfolge oder es gilt $k = n$ (bzw. $k = 0$).*

Beweis: Betrachte $\sigma(1, i)$ als Funktion von $i \in [0 : n]$ wie in Abbildung 1.20 dargestellt.

Allgemein gilt $\sigma(i, j) = \sigma(1, j) - \sigma(1, i - 1)$.

Fall 1: Position $k > 0$ befindet sich in keiner maximal bewerteten Teilfolge. Da der folgende Wert a_{k+1} (sofern vorhanden) aufgrund der Definition von k positiv sein muss, ist nach Korollar 1.8 die Position k am Ende einer nicht maximal bewerteten

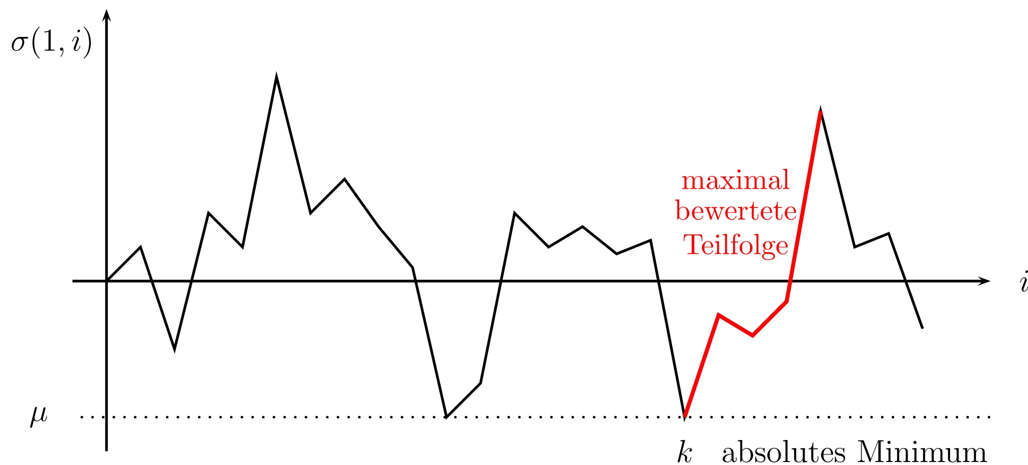


Abbildung 1.20: Skizze: Die Funktion $\sigma(1,i)$

Teilfolge. Also ist entweder $k = n$ oder an Position $k+1$ beginnt eine maximal bewertete Teilfolge.

Fall 2: Position $k > 0$ befindet sich in einer maximal bewerteten Teilfolge a'. Angenommen a' beginnt an Position $i \leq k$. Dann gilt aber nach Definition von k, dass $\sigma(i,k) \leq 0$ ist, was im Widerspruch zu Lemma 1.5 Charakterisierung 3 steht und Fall 2 kann gar nicht eintreten.

Fall 3: Falls die Position $k = 0$ ist, dann ist a_1 positiv und befindet sich nach Korollar 1.8 in einer maximal bewerteten Teilfolge und muss darin an erster Position sein.

Den Beweis für das Maximum von $\sigma(1,i)$ sei dem Leser überlassen. ■

Wir werden von nun an die folgende algorithmische Idee verfolgen: Sei a' eine Teilfolge von a, die die Eigenschaft E1 erfüllt. Wir werden versuchen a' zu einer maximal bewerteten Teilfolge zu verlängern.

Definition 1.11 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine Folge reeller Zahlen. Eine Teilfolge a' von a heißt a-MSS, wenn $a' \in \mathrm{MSS}(a)$.*

Lemma 1.12 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Sei a'' eine Teilfolge von a' und a' eine Teilfolge von a. Ist a'' eine a-MSS, dann ist a'' auch a'-MSS.*

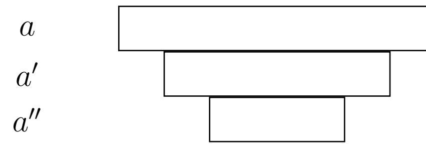


Abbildung 1.21: Skizze: a'' ist a-MSS

Beweis: Sei a'' eine a-MSS und sei sowohl a' eine Teilfolge von a als auch eine Oberfolge von a'', wie in Abbildung 1.21 dargestellt.

Da a'' eine a-MSS ist, erfüllt a'' die Eigenschaften E1 und E2 bezüglich a. Somit erfüllt die Folge a'' diese Eigenschaften auch bezüglich der Oberfolge a'. ■

Lemma 1.13 *Sei* $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. *Sei* $a' = (a_1, \ldots, a_i)$ *und* $a_k \leq 0$ *für alle* $k \in [i+1 : n]$. *Alle maximal bewerteten Teilfolgen von* a' *sind auch maximal bewertete Teilfolgen von* a *und umgekehrt.*

Beweis: Wir beweisen beide Implikationen getrennt.

⇐: Dies ist die Aussage von Lemma 1.12, da nach Lemma 1.5 Charakterisierung 3 jede a-MSS eine Teilfolge von a' sein muss.

⇒: Sei a'' eine a'-MSS, wie in Abbildung 1.22 illustriert. Wir führen den Beweis

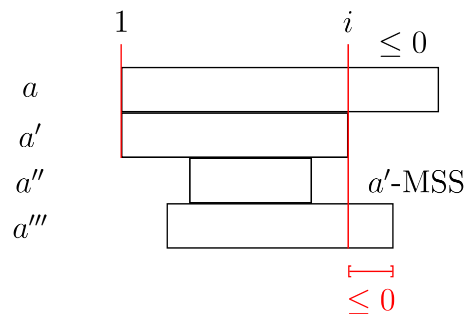


Abbildung 1.22: Skizze: a'' ist eine a'-MSS

durch Widerspruch und nehmen daher an, dass a'' keine a-MSS ist. Dann muss es eine Oberfolge a''' von a'' geben, die E1 erfüllt. Da a'' eine a'-MSS ist, muss diese Oberfolge a''' in den hinteren Teil (ab Position $i + 1$) hineinragen. Eine solche Folge a''' kann nach Lemma 1.5 Charakterisierung 3 nicht E1 erfüllen, da der Suffix von a''' ab Position $i + 1$ einen nichtpositiven Score besitzt. Dies führt zu dem gewünschten Widerspruch. ■

Lemma 1.14 *Sei* $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$*. Sei* $a' = (a_i, \ldots, a_j)$ *eine* a*-MSS und sei* $a^L = (a_1, \ldots, a_{i-1})$ *und* $a^R = (a_{j+1}, \ldots, a_n)$*. Dann ist eine Teilfolge* a'' *von* a^L *(bzw.* a^R*) genau dann eine* a^L*-MSS (bzw.* a^R*-MSS), wenn* $a'' \neq a'$ *eine* a*-MSS ist.*

Beweis: Wir beweisen beide Implikationen getrennt.

⇐: Nach Lemma 1.6 sind maximal bewertete Teilfolgen disjunkt. Somit ist jede andere maximal bewertete Teilfolge eine Teilfolge von a^L oder a^R. Mit Lemma 1.12 ist nun jede maximal bewertete Teilfolge von a ungleich a' entweder eine maximal bewertete Teilfolge von a^L oder a^R.

⇒: Wir führen den Beweis durch Widerspruch. Sei dazu a'' eine a^L-MSS, aber keine a-MSS. Somit erfüllt a'' die Eigenschaft E1. Mit Lemma 1.7 folgt, dass es eine Oberfolge $\overline{a}$ von a'' gibt, die eine a-MSS ist.

Angenommen $\overline{a}$ wäre eine Teilfolge von a^L. Dann wäre nach Lemma 1.12 $\overline{a}$ auch eine a^L-MSS. Somit wären $\overline{a}$ und a'' überlappende a^L-MSS, was Lemma 1.6 widerspricht.

Somit müssen sich $\overline{a}$ und a' überlappen. Da aber beide a-MSS sind, ist dies ebenfalls ein Widerspruch zu Lemma 1.6. ■

Damit können wir nun die Äquivalenz der strukturellen und prozeduralen Definition maximal bewerteter Teilfolgen zeigen.

Theorem 1.15 *Die strukturelle und prozedurale Definition von maximal bewerteten Teilfolgen stimmen überein.*

Beweisidee: Sei MSS(a) die Menge aller Teilfolgen von a, die maximal bewertete Teilfolgen sind (gemäß der strukturellen Definition, also die die Eigenschaften E1 und E2 erfüllen).

Sei a' eine kürzeste Teilfolge mit maximalem Score. Man überlegt sich leicht, dass diese in a maximal bewertet ist, d.h. dass a' die Eigenschaften E1 und E2 erfüllt und somit $a' = (a_i, \ldots, a_j) \in \text{MSS}(a)$. Nach Lemma 1.14 gilt:

$$\text{MSS}(a) = \{a'\} \cup \text{MSS}(a_1, \ldots, a_{i-1}) \cup \text{MSS}(a_{j+1}, \ldots, a_n).$$

Der vollständige Beweis lässt sich jetzt formal mittels vollständiger Induktion führen. ■

1.2.3 Ein Algorithmus zur Lösung

Wir beschreiben jetzt einen Algorithmus zur Ermittlung aller maximal bewerteten Teilfolgen.

- Die Eingabe ist die Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$.
- Die Elemente werden von links nach rechts verarbeitet. Wir betrachten also der Reihe nach jedes Präfixes $a' = (a_1, \ldots, a_m)$ mit $m \in [0 : n]$.
- Dabei merken wir uns disjunkte Teilfolgen $I_1, \ldots, I_k$ eines Präfixes a' von a, die maximal bewertete Teilfolgen des bereits abgearbeiteten Präfixes a' sein werden.
- $I_i = (a_{\ell_i}, \ldots, a_{r_i})$, d.h. $I_i \hat{=} (\ell_i, r_i)$.
- Setze $L_i = \sigma(1, \ell_i - 1)$ und $R_i = \sigma(1, r_i)$.
- Damit gilt $\sigma(I_i) := \sigma(\ell_i, r_i) = R_i - L_i$.

Zu Beginn ist a' die leere Folge und wir setzen $m := 0$ und $k := 0$ und merken uns dazu $S = \sigma(1, m)$ (also zu Beginn $S := 0$) und aktualisieren $S := S + a_m$ bei einer Erhöhung von m. Die Werte von S können vor bzw. nach der Aktualisierung verwendet werden, um ggf. L_i bzw. R_i zu bestimmen.

Das können wir uns wie in Abbildung 1.23 veranschaulichen.

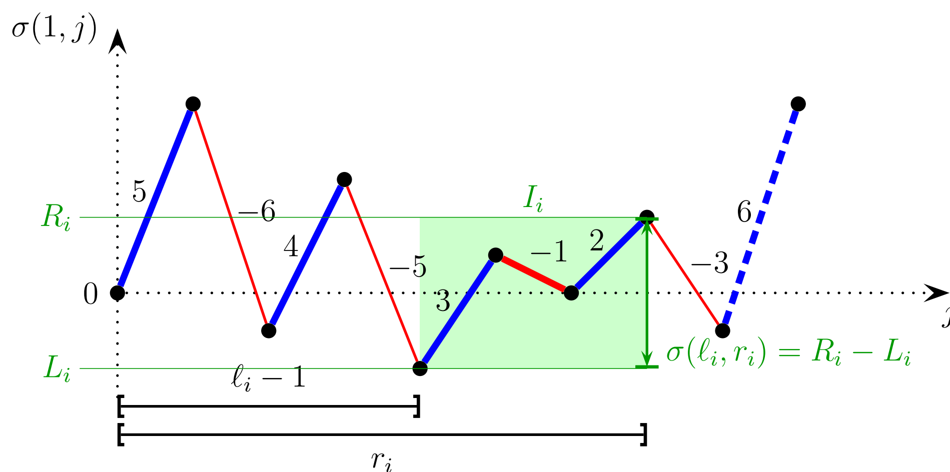


Abbildung 1.23: Skizze: Die Werte ℓ_i, r_i, L_i, R_i und I_i

Wir gehen wie folgt vor: Bearbeite die Folge von links nach rechts und sammle eine Liste von maximal bewerteten Teilfolgen bezüglich des betrachteten Präfixes.

Sei a_m das aktuell betrachtete Element der Folge a (nach einer Erhöhung von m):

- Ist $a_m \leq 0$, betrachte die nächste Position $m + 1$, da keine maximal bewertete Teilfolge an Position m beginnen oder enden kann.
- Ist $a_m > 0$, inkrementiere k, erzeuge eine neue einelementige Liste $I_k = (m, m)$ und bestimme L_k sowie R_k (mithilfe von S). Die Folge (a_m) erfüllt E1, aber nicht notwendigerweise E2 im betrachteten Präfix $a' = (a_1, \ldots, a_m)$ von a.
- Für ein neues I_k wiederhole das Folgende: Durchsuche $I_{k-1}, \ldots, I_1$ (von rechts nach links!) bis ein maximales j gefunden wird, so dass $L_j < L_k$ und betrachte die folgenden drei Fälle genauer. Hierbei gilt immer, dass $I_1, \ldots, I_{k-1}$ maximal bewertete Teilfolgen in $(a_1, \ldots, a_{\ell_k - 1})$ sind und I_k zumindest die Eigenschaft E1 erfüllt.

  Fall 1: Es existiert gar kein solches j, siehe Abbildung 1.24.

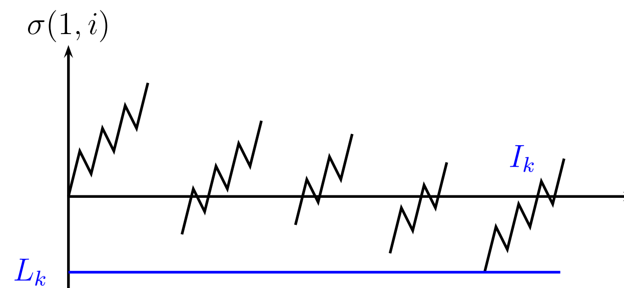
  

  Abbildung 1.24: Skizze: Nichtexistenz von j

  Nach Lemma 1.10 ist I_k der Anfang einer maximal bewerteten Teilfolge in der Teilfolge $a' = (a_1, \ldots, a_m)$. Nach Lemma 1.14 sind $I_1, \ldots, I_{k-1}$ dann auch maximal bewertete Teilfolgen in a'.

  Nach Lemma 1.7 ist I_k auch Teilfolge einer a-MSS a''. Die Teilfolge I_k ist sogar Anfang von a'', denn sonst hätten einige Präfixe von a'' einen nichtpositiven Score, was ein Widerspruch zu Lemma 1.5 ist. Somit sind $I_1, \ldots, I_{k-1}$ nach Lemma 1.14 auch jeweils eine a-MSS.

  Im Algorithmus geben wir daher die Teilfolgen $I_1, \ldots, I_{k-1}$ als maximal bewerteten Teilfolgen aus und setzen $I_1 := I_k$, $(\ell_1, r_1) = (\ell_k, r_k)$, $(L_1, R_1) := (L_k, R_k)$ sowie $k := 1$. □

  Fall 2: Sei j maximal mit $L_j < L_k$ und $R_j < R_k$, siehe Abbildung 1.25.

  Offensichtlich gilt nach Wahl von j, dass $L_i \geq L_k$ für $i \in [j + 1 : k - 1]$. Weiterhin gilt $R_i \leq R_j$ für alle $i \in [j + 1 : k - 1]$. Andernfalls gäbe es ein $i \in [r_j + 1 : \ell_k - 1]$ mit $\sigma(1, i) > R_j$. Sei i ein minimaler solcher Index.

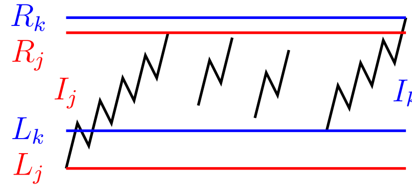


Abbildung 1.25: j maximal mit $L_j < L_k$ und $R_j < R_k$

Dann besäße I_j die Oberfolge $(a_{\ell_j}, \ldots, a_i)$, die offensichtlich die Eigenschaft E1 erfüllt, was der Eigenschaft E2 von I_j in $(a_1, \ldots, a_{\ell_k - 1})$ widerspricht. Beachte auch, dass wegen Korollar 1.8 alle Elemente außerhalb der Teilfolge, $I_1, \ldots, I_k$ nichtpositiv sein müssen.

Somit erfüllt die Teilfolge $(a_{\ell_j}, \ldots, a_{r_k})$ die Eigenschaft E1, aber nicht notwendigerweise E2. Somit muss ein neue Teilfolge generiert und angefügt werden. Wir bilden aus den Teilfolgen $I_j, \ldots, I_k$ eine neue Teilfolge, d.h. wir setzen $I_j := (\ell_j, r_k)$, $r_j := r_k$, $R_j := R_k$ sowie $k := j$ und wiederholen die Prozedur für $I_k = I_j$ (also mit dem neuen k). Die alten Teilfolgen $I_{j+1}, \ldots, I_k$ müssen dabei natürlich gelöscht werden.

Alle 3 Fälle sind jetzt wieder neu zu betrachten. □

Fall 3: Sei j maximal mit $L_j < L_k$ und $R_j \geq R_k$, siehe Abbildung 1.26.

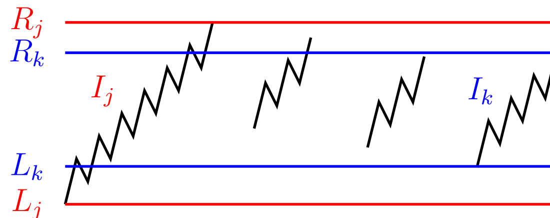


Abbildung 1.26: Skizze: j maximal mit $L_j < L_k$ und $R_j \geq R_k$

Wir zeigen zunächst, dass dann I_j eine a'-MSS mit $a' = (a_1, \ldots, a_m)$ ist.

E1) Dies gilt nach Konstruktion, da I_j $(j < k)$ bereits eine maximal bewertete Teilfolge von $(a_1, \ldots, a_{\ell_k - 1})$ ist und somit die Eigenschaft E1 gilt.

E2) Für einen Widerspruchsbeweis nehmen wir an, es gäbe eine Oberfolge $a'' = (a_i, \ldots, a_s)$ von I_j, die die Eigenschaft E1 erfüllt.

Gilt $r_j \leq s < \ell_k \leq m$, dann wäre a'' auch eine Oberfolge von I_j, die in $(a_1, \ldots, a_{\ell_k - 1})$ liegt, und somit wäre I_j keine maximal bewertete Teilfolge von $(a_1, \ldots, a_{\ell_k - 1})$ gewesen, was aufgrund unserer Konstruktion ein Widerspruch ist.

Also muss $s \in [\ell_k : r_k]$ gelten. Dann gibt es aber offensichtlich Suffixe von a'' mit nichtpositiven Score, was nach Lemma 1.5 Charakterisierung 3) nicht sein kann.

Jetzt zeigen wir noch, dass auch $I_k = (a_{\ell_k}, \ldots, a_{r_k})$ eine maximal bewertete Teilfolge von $a' = (a_1, \ldots, a_m)$ ist. Da I_j eine maximal bewertete Teilfolge von a' ist genügt es nach Lemma 1.14 zu zeigen, dass I_k eine maximal bewertete Teilfolge von $a'' = (a_{r_j+1}, \ldots, a_m)$ ist. Aus Lemma 1.10 folgt aber unmittelbar, dass an Position $\ell_k \leq m$ eine maximal bewertete Teilfolge beginnt. Da jeder echte Präfix von I_k die Folgen I_k als Oberfolge besitzt und diese die Eigenschaft E1 erfüllt, kann kein echter Präfix von I_k maximal bewertet sein. Also muss I_k selbst eine a''-MSS sein und die Behauptung gilt.

In diesem Fall ist algorithmisch also gar nichts zu tun. □

In jedem Fall wird das Durchsuchen der Listen $I_{k-1}, \ldots, I_1$ entweder erneut aufgerufen oder aber alle aufgesammelten Teilfolgen $I_1, \ldots, I_k$ sind maximal bewertete Teilfolgen des Präfixes $a' = (a_1, \ldots, a_m)$ von a (die bereits ausgegebenen maximal bewerteten Folgen von a ignorieren wir einfach).

In Abbildung 1.27 auf Seite 24 ist ein Ablauf des gerade beschriebenen AMSS-Algorithmus für die Sequenz $a = (+3, -2, +3, -3, +2, -4, +3)$ schematisch dargestellt. Hierbei entsprechen die roten Linienzüge den gespeicherten disjunkten Teilfolgen $I_1, \ldots, I_k$, während die grünen Linienzüge die nach Fall 1 ausgegeben maximal bewerteten Teilfolgen darstellen.

1.2.4 Zeitkomplexität

Wir analysieren jetzt die Zeitkomplexität des soeben vorgestellen Algorithmus:

1.) Die Betrachtung von a_m und der eventuell Generierung des neuen I_k kann pro Folgenelement in konstanter Zeit durchgeführt werden. Daraus ergibt sich ein Gesamtaufwand von $O(n)$.

2.) Das Durchsuchen der bislang gesammelten Teilfolgen $I_{k-1}, \ldots, I_1$ wird anschließend analysiert.

3.) Gesamtaufwand von Fall 1: Es existiert kein j, dann geben wir $I_1, \ldots, I_{k-1}$ aus. Dies lässt sich für jedes I_ℓ in konstanter Zeit erledigen, also erhalten wir insgesamt für alle Ausgaben einen Zeitbedarf von $O(n)$. Es kann nur $O(n)$ maximal bewertete Teilfolgen geben, da diese ja nach Lemma 1.6 disjunkt sind.

4.) Gesamtaufwand von Fall 3: Es gilt $L_j < L_k \wedge R_j \geq R_k$. Die Ermittlung des dritten Falles geschieht in konstanter Zeit. Da anschließend das Durchsuchen

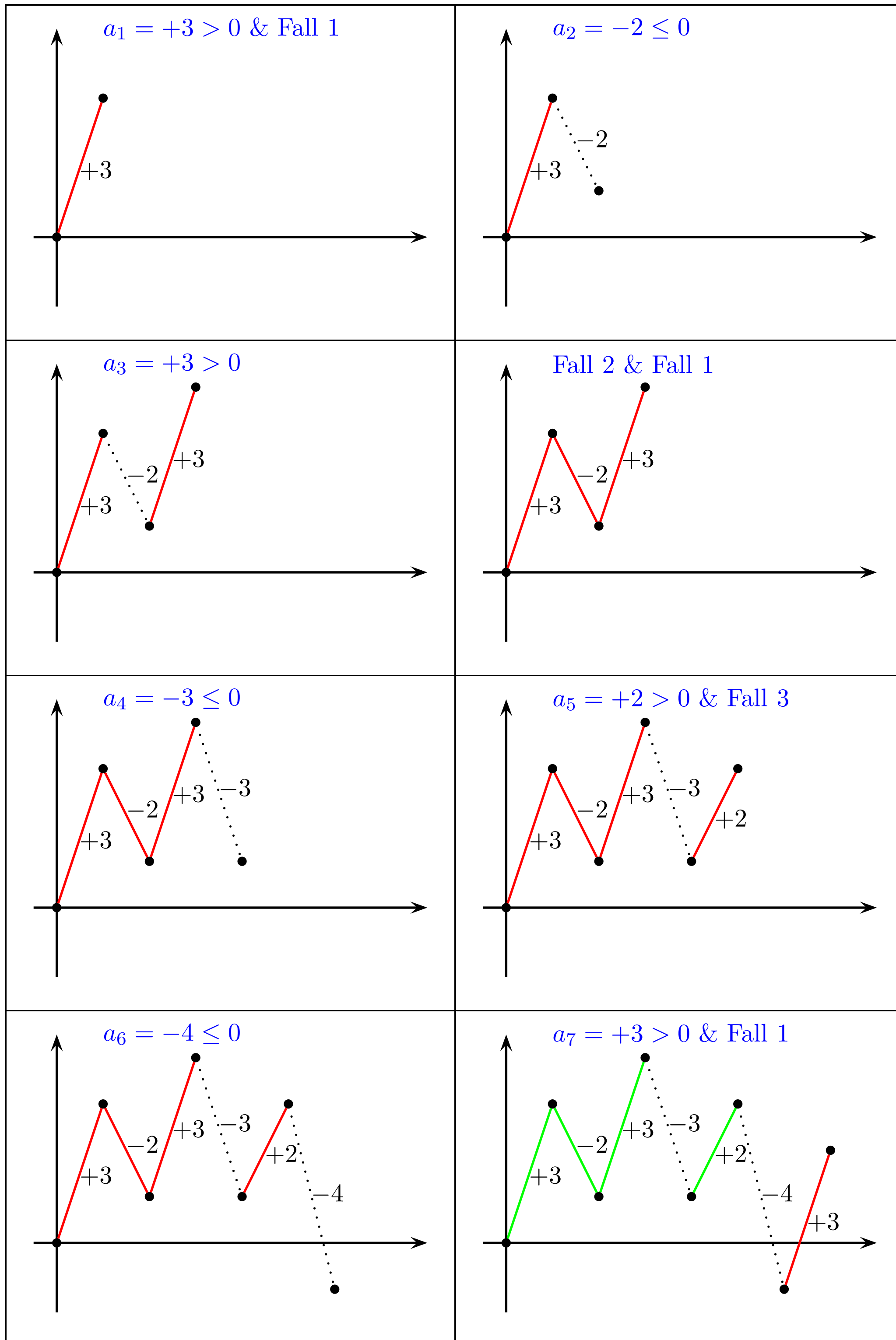


Abbildung 1.27: Beispiel: $a = (+3, -2, +3, -3, +2, -4, +3)$

beendet ist, tritt dieser Fall maximal n-mal auf, also ist der Aufwand für alle diese Fälle höchstens $O(n)$.

5.) Gesamtaufwand von Fall 2: Es gilt $L_j < L_k \wedge R_j < R_k$. Die Verschmelzung von $I_j, \ldots, I_k$ lässt sich in Zeit $O(k - j + 1)$ erledigen (explizites Löschen der Teilfolgen). Da insgesamt höchstens $n - 1$ Verschmelzungen von einzelnen Teilfolgen möglich sind (spätestens dann müssen sich alle Folgenglieder in einer einzigen Folge befinden), beträgt der Gesamtzeitbedarf $O(n)$.

Dabei stellen wir uns eine Verschmelzung von ℓ Teilfolgen als $\ell - 1$ Verschmelzungen von je zwei Teilfolgen vor. Da es insgesamt maximal $n - 1$ Verschmelzungen disjunkter Teilfolgen geben kann, folgt obige Behauptung.

Durchsuchen der Listen: Beim Durchsuchen der Listen von Teilfolgen gehen wir etwas geschickter vor. Wir merken uns für jede Teilfolge I_k dabei die Teilfolge I_j, für die $L_j < L_k$ und j maximal ist. Beim nächsten Durchlaufen können wir dann einige Teilfolgen beim Suchen einfach überspringen. Dies ist in der folgenden Abbildung 1.28 schematisch dargestellt.

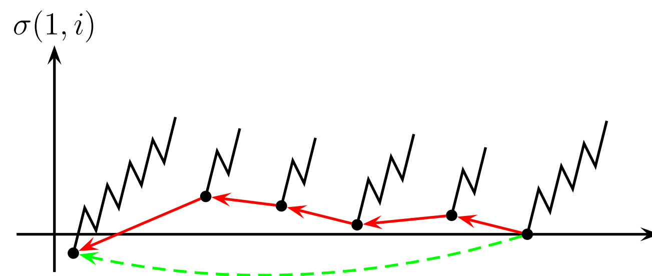


Abbildung 1.28: Skizze: Übersprungene Folge von L_j-Werten beim Durchsuchen

Jetzt müssen wir uns nur noch überlegen, welchen Aufwand alle Durchsuche-Operationen insgesamt haben. Dabei werden die Kosten zum einen auf die Aufrufe (der Durchsucheprozedur) und zum anderen auf die ausgeschlossenen Teilfolgen verteilt.

Bei einem Durchsuchen der Liste werden beispielsweise ℓ Teilfolgen übersprungen. Dann wurden $\ell+2$ Teilfolgen inspiziert (und ℓ Teilfolgen ausgeschlossen), was Kosten in Höhe von $O(\ell+1)$ verursacht (man beachte, dass $\ell \geq 0$ gilt). Die Kosten in Höhe von $O(\ell)$ werden anteilig auf die ausgeschlossenen Teilfolgen umgelegt. Somit erhält jedes ausgeschlossene Teilfolge Kosten von $O(1)$. Die Kosten für den Rest werden auf den Aufruf eines Durchsuch-Durchlaufs umgelegt, die dann ebenfalls konstant sind. Somit fallen jeweils Kosten $O(1)$ für jede ausgeschlossene Teilfolge und jeden Aufruf an.

Anzahl Aufrufe: Es kann maximal so viele Aufrufe geben, wie der Zeitbedarf in der Analyse der Punkte 3, 4 und 5 verbraucht wird, da ja nach jedem Durch-

suchen entweder Fall 1, Fall 2 oder Fall 3 ausgeführt wird (die mindestens konstante Kosten verursachen). Also ist die Anzahl Aufrufe $O(n)$.

Anzahl ausgeschlossener Teilfolgen: Zum einen können diese im Fall 1 ausgegeben werden, von diesen kann es daher ebenfalls maximal $O(n)$ viele geben.

Wenn diese Teilfolgen nicht selbst ausgegeben werden, müssen diese irgendwann mit anderen Teilfolgen verschmolzen worden sein (sie können ja nicht einfach verschwinden). Da, wie wir bereits gesehen haben, maximal $O(n)$ Teilfolgen verschmolzen werden, können auch hierdurch höchstens $O(n)$ Teilfolgen ausgeschlossen werden.

Mit unserem kleinen Trick kann also auch das Durchsuchen der Listen von Teilfolgen mit einem Aufwand von insgesamt $O(n)$ bewerkstelligt werden. Halten wir das Resultat noch im folgenden Satz fest.

Theorem 1.16 *Das All Maximal Scoring Subsequences Problem für eine gegebene reelle Folge kann in Linearzeit gelöst werden.*

1.3 Bounded All Maximum Scoring Subsequences (*)

Im vorherigen Abschnitt haben wir quasi über eine lokale Definition die Menge aller maximal bewerteten Teilfolgen bestimmt. Wir können auch einen globalen Ansatz wählen und eine Menge von Teilfolgen einer gegebenen Folge fordern, deren aufaddierte Scores maximal ist. Dies ist jedoch nicht sinnvoll, da dann die Menge aller einelementigen Teilfolgen, die positive Elemente beschreiben, eine Lösung dieses Problems ist. Man überlegt sich leicht, dass dies wirklich ein globales Optimum ist.

Daher wollen wir in diesem Abschnitt das Problem ein wenig abwandeln, in dem wir die Länge der Teilfolgen in der Lösungsmenge sowohl von oben als auch von unten beschränken.

1.3.1 Problemstellung

Es wird zusätzlich eine untere $\underline{\lambda}$ und eine obere Schranke $\overline{\lambda}$ vorgegeben, um die Länge der zu betrachtenden Teilfolgen zu beschränken, wobei natürlich $\underline{\lambda} \leq \overline{\lambda} \in \mathbb{N}$ gilt. Mit $Seq(n, k)$ bezeichnen wir die Menge aller 0-1-Zeichenreihen mit genau k konsekutiven 1-Runs, deren Länge durch $\underline{\lambda}$ nach unten und mit $\overline{\lambda}$ nach oben beschränkt ist.

Notation 1.17 *Sei $k \leq n \in \mathbb{N}$ und $\underline{\lambda} \leq \overline{\lambda} \in \mathbb{N}$, dann bezeichne*

$$Seq(n, k) \ := \ \{0^*1^{m_1}0^+1^{m_2}0^+ \cdots 0^+1^{m_k}0^* \mid \underline{\lambda} \leq m_i \leq \overline{\lambda}\} \cap \{0,1\}^n \ \subseteq \ \{0,1\}^n.$$

Bounded All Maximum Scoring Subsequences (BAMSS)

Eingabe: Eine Folge $(a_1, \ldots, a_n) \in \mathbb{R}^n$, $\underline{\lambda} \leq \overline{\lambda} \in \mathbb{N}$.
Gesucht: Eine Sequenz $s \in \bigcup_{k=0}^{n} Seq(n,k)$, die $\sum_{i=1}^{n} s_i \cdot a_i$ maximiert.

Bemerkung: Durch die Längenbeschränkung können nach unserer alten Definition von allen Maximal Scoring Subsequences die einzelnen Teilfolgen der Lösung überlappen, siehe Abbildung 1.29. Damit ist ein Greedy-Ansatz nicht mehr effizient

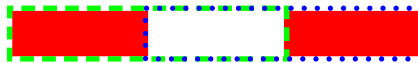


Abbildung 1.29: Skizze: Erlaubte überlappende Sequenzen

möglich. Aus diesem Grund wurde das Problem anders gefasst, nämlich wie oben. Mit Hilfe der Menge $Seq(n,k)$ werden aus der Gesamtfolge Teilstücke ausgewählt (nämlich 1-Runs), die wir als Lösungsteilfolgen zulassen wollen.

Weiterhin sollte man beachten, dass für eine gegeben der Folge der Länge n auch für $\underline{\lambda} = 1$ und $\overline{\lambda} = n$ die Lösungen für das BAMSS und AMSS verschieden sein können. Betrachte hierzu die Folge $a = (+3, -2, +3)$. Die eindeutige Lösung für AMSS lautet $\{(a_1, a_2, a_3)\}$ und für BAMSS $\{(a_1), (a_3)\}$.

1.3.2 Lösung mittels Dynamischer Programmierung

Wir definieren wieder eine Tabelle S für $i, k \in [0 : n]$ wie folgt:

$$S(i,k) := \max\left\{\sum_{j=1}^{i} s_j \cdot a_j \ : \ s \in Seq(i,k)\right\}$$

Wir müssen also $S(i,k)$ für alle Werte $i \in [0 : n]$, $k \in [0 : n]$ berechnen, wobei $\max \emptyset = \max\{\} = -\infty$ (entspricht dem neutralem Element für das Minimum) gilt.

Für diese Tabelle S lässt sich die folgende Rekursionsgleichung aufstellen:

$$\begin{aligned}
S(i,0) &= 0 \quad \text{für } i \in [-1 : n], \\
S(i,k) &= -\infty \quad \text{für } i < k \cdot \underline{\lambda} + (k-1), \\
S(i,k) &= \max\left\{ \begin{array}{c} S(i-1,k), \\ S(i-\lambda-1, k-1) + \sum\limits_{j=i-\lambda+1}^{i} a_j \end{array} \ : \ \lambda \in [\underline{\lambda} : \min(\overline{\lambda}, i)] \right\}.
\end{aligned}$$

Die Korrektheit der Rekursionsgleichung ergibt sich aus der Tatsache, dass man sich eine optimale Menge von Teilfolgen anschaut und unterscheidet, ob diese mit einer 0 oder einem 1-Run endet, wie in Abbildung 1.30 illustriert.

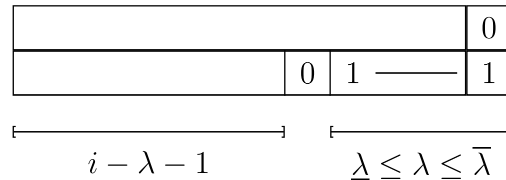


Abbildung 1.30: Skizze: Aufstellen der Rekursionsgleichung

Wie man leicht sieht, gilt:

$$\sum_{j=i-\lambda+1}^{i} a_j = \sum_{j=1}^{i} a_j - \sum_{j=1}^{i-\lambda} a_j$$

Somit kann man die einzelnen Summen effizient berechnen, wenn für alle $i \in [0:n]$ die folgenden Summen bekannt sind: $A(i) := \sum_{j=1}^{i} a_j$. Berechnet man die Werte $A(i)$ für $i \in [0:n]$ vorab, so lässt sich jeder Eintrag der Tabelle S in Zeit $O(\overline{\lambda} - \underline{\lambda}) = O(n)$ berechnen. Falls die Summen zu groß werden sollten, kann man diese auch für einige jeweils kleinere Teilbereiche ermitteln.

Die Gesamtlaufzeit beträgt somit $O(n^2/\underline{\lambda}(\overline{\lambda} - \underline{\lambda})) \leq O(n^2(\overline{\lambda} - \underline{\lambda})) \leq O(n^3)$. In der Praxis sind die Schranken $\underline{\lambda}$ und $\overline{\lambda}$ allerdings so nah beieinander, so dass $\overline{\lambda} - \underline{\lambda} = O(1)$ gilt und somit die Laufzeit bei $O((\overline{\lambda} - \underline{\lambda})n^2) = O(n^2)$ bleibt. Gilt $(\overline{\lambda} - \underline{\lambda}) = O(\underline{\lambda})$, so bleibt es auch bei einer Laufzeit von $O(n^2)$.

Theorem 1.18 *Das BAMSS-Problem lässt sich in Zeit $O(n^2/\underline{\lambda} \cdot (\overline{\lambda} - \underline{\lambda}))$ lösen.*

Wie wir im nächsten Abschnitt sehen werden, können wir das Problem auch effizienter lösen. Die vorhergehende Lösung hat den Vorteil dass man auch die Maximalanzahl der Teilfolgen einer Lösung beschränken kann, was durchaus biologisch sinnvoll sein kann.

1.3.3 Effiziente Lösung mittels Dynamischer Programmierung

Wir definieren jetzt folgende Tabelle S für $i \in [0:n]$ wie folgt:

$$S(i) := \max\left\{ \sum_{j=1}^{i} s_j \cdot a_j \ : \ s \in Seq(i,k) \wedge k \in [0:n] \right\}$$

Wir müssen also in diesem Fall $S(i)$ für alle Werte $i \in [0 : n]$ berechnen, wobei $\max \emptyset = \max\{\} = -\infty$ (entspricht dem neutralem Element) gilt.

Für diese Tabelle S lässt sich die folgende Rekursionsgleichung aufstellen:

$$\begin{aligned} S(i) &= 0 \quad \text{für } i \in [-1 : \underline{\lambda} - 1], \\ S(i) &= \max \left\{ \begin{array}{c} S(i-1), \\ S(i-\lambda-1) + \sum_{j=i-\lambda+1}^{i} a_j \end{array} : \lambda \in [\underline{\lambda} : \min\{i, \overline{\lambda}\}] \right\}. \end{aligned}$$

Die Korrektheit der Rekursionsgleichung ergibt sich wie vorher. Für eine einfachere Angabe der Rekursionsgleichung benötigen wir auch hier noch die Definition $S(-1) = 0$.

Die Gesamtlaufzeit beträgt somit $O(n(\overline{\lambda} - \underline{\lambda})) \leq O(n^2)$. In der Praxis sind die Schranken $\underline{\lambda}$ und $\overline{\lambda}$ allerdings so nah beieinander, so dass $\overline{\lambda} - \underline{\lambda} = O(1)$ gilt und somit die Laufzeit bei $O((\overline{\lambda} - \underline{\lambda})n) = O(n)$ bleibt.

Theorem 1.19 *Das BAMSS-Problem lässt sich in Zeit $O(n(\overline{\lambda} - \underline{\lambda}))$ lösen.*

1.4 Bounded Maximal Scoring Subsequence (*)

Jetzt wollen wir nur *eine* längenbeschränkte Maximal Scoring Subsequence finden. Man beachte, dass diese kein Teil der Lösung aus dem Problem des vorherigen Abschnittes sein muss! Wir werden zunächst nur eine obere Längenbeschränkung für die gesuchte Folge betrachten. Eine Hinzunahme einer unteren Längenbeschränkung ist nicht weiter schwierig und wird in den Übungen behandelt.

1.4.1 Problemstellung

Wir formalisieren zunächst die Problemstellung.

Bounded Maximal Scoring Subsequence (BMSS)

Eingabe: Eine Folge $(a_1, \ldots, a_n) \in \mathbb{R}^n$ und $\lambda \in \mathbb{N}$.
Gesucht: Eine (zusammenhängende) Teilfolge $(a_i, \ldots, a_j)$ mit $j - i + 1 \leq \lambda$, die $\sigma(i,j)$ maximiert, wobei $\sigma(i,j) = \sum_{\ell=i}^{j} a_\ell$.

1.4.2 Links-Negativität

Um einer effizienten Lösung des Problems näher zu kommen, benötigen wir zuerst den Begriff der Links-Negativität und einer minimalen linksnegativen Partition einer reellen Folge.

Definition 1.20 *Eine Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ reeller Zahlen heißt* linksnegativ, *wenn $\sum_{\ell=1}^{k} a_\ell \leq 0$ für alle $k \in [1 : n-1]$.*

Eine Partition $a = A_1 \cdots A_k$ der Folge a heißt minimal linksnegativ, *wenn für alle $i \in [1 : k]$ A_i linksnegativ ist und $\sigma(A_i) > 0$ für alle $i \in [1 : k-1]$ ist.*

Beispiele:

1.) $(-1, 1, -3, 1, 1, 3)$ ist linksnegativ,

2.) $(2, -3, -4, 5, 3, -3)$ ist **nicht** linksnegativ.
Die Partition $(2)(-3, -4, 5, 3)(-3)$ ist eine minimal linksnegative.

Lemma 1.21 *Jede Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ lässt sich eindeutig in eine minimal linksnegative Partition zerlegen.*

Beweis: Wir führen den Beweis durch Induktion über n.

Induktionsanfang ($n = 1$): Dies folgt unmittelbar aus der Definition.

Induktionsschritt ($n \to n+1$): Betrachte $a' = (a_0, a_1, \ldots, a_n)$ und sei dann $a = (a_1, \ldots, a_n)$. Nach Induktionsvoraussetzung sei $a = A_1 \cdots A_k$ die(!) minimal linksnegative Partition von a.

Fall 1 ($a_0 > 0$): Indem wir a_0 einfach als Segment an die minimal linksnegative Partition von a voranstellen, erhalten wir eine minimal linksnegative Partition für $a' = (a_0) \cdot A_1 \cdots A_k$.

Fall 2 ($a_0 \leq 0$): Wähle ein minimales i mit

$$a_0 + \sum_{j=1}^{i} \sigma(A_j) > 0.$$

Dann ist $((a_0) \cdot A_1 \cdots A_i) \cdot A_{i+1} \cdots A_k$ eine minimal linksnegative Partition. Hierzu genügt es zu zeigen, dass $(a_0) \cdot A_1 \cdots A_i$ linksnegativ ist. Nach Konstruktion gilt

$\sigma((a_0) \cdot A_1 \cdots A_j) \leq 0$ für $j \in [1 : i-1]$. Auch für jedes echte Präfix $(a_0) \cdot A_1 \cdots A_{j-1} \cdot A'_j$ davon muss $\sigma((a_0) \cdot A_1 \cdots A_{j-1} \cdot A'_j) \leq 0$ sein, da sonst bereits $\sigma(A'_j) > 0$ gewesen sein müsste. Dies kann nicht der Fall sein, da A_j linksnegativ ist, weil $A_1 \cdots A_k$ eine minimale linksnegative Partition von a ist.

Der Beweis der Eindeutigkeit sei dem Leser zur Übung überlassen. ■

Im Folgenden ist die Notation für die minimale linksnegative Partition der Suffixe der untersuchten Folge hilfreich.

Notation 1.22 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Jedes Suffix $a^{(i)} = (a_i, \ldots, a_n)$ von a besitzt dann die eindeutige minimal linksnegative Partition $a^{(i)} = A_1^{(i)} \cdots A_{k_i}^{(i)}$.*

Basierend auf dieser Notation lassen sich linksnegative Zeiger definieren.

Definition 1.23 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge und für jedes Suffix ab Position i sei $a^{(i)} = A_1^{(i)} \ldots A_{k_i}^{(i)}$ die zugehörige minimale linksnegative Partition. Sei weiter $A_1^{(i)} = (a_i, \ldots, a_{p(i)})$, dann heißt $p(i)$ der* linksnegative Zeiger *von i.*

Bemerkungen:

- Ist $a_i > 0$, dann ist $p(i) = i$.
- Ist $a_i \leq 0$, dann ist $p(i) > i$ für $i < n$ und $p(n) = n$.

1.4.3 Algorithmus zur Lösung des BMSS-Problems

Wie helfen uns minimale linksnegative Partitionen bei der Lösung des Problems? Wenn wir eine Teilfolge $a' = (a_i, \ldots, a_j)$ als Kandidaten für eine längenbeschränkte MSS gefunden haben, dann erhalten wir den nächstlängeren Kandidaten durch das

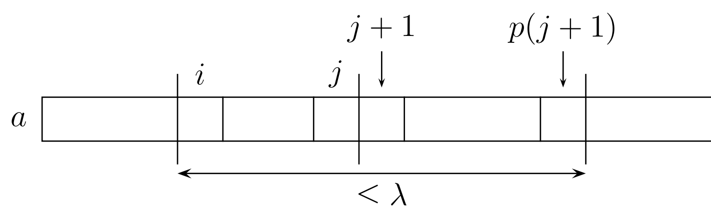


Abbildung 1.31: Skizze: Verlängerung einer maximal Scoring Subsequence

```
BMSS (real a[], int n, λ)
begin
    // score and position of current mss
    int ms := 0,   mi := 1,   mj := 0;
    int i := 1,   j := 0 ;  /* (a_i, ..., a_j) is current candidate subsequence */
    for (i := 1; i ≤ n; i++) do
        while ((a_i ≤ 0) && (i < n)) do
            i++;
        j := max(i, j);
        while ((j < n) && (p[j + 1] < i + λ) && (σ(j + 1, p(j + 1)) > 0)) do
            j := p(j + 1);
            if (σ(i, j) > ms) then
                mi := i;
                mj := j;
                ms := σ(i, j);
end
```

Abbildung 1.32: Algorithmus: Lösung das BMSS Problems

Anhängen der Teilfolge $(a_{j+1}, \ldots, a_{p(j+1)})$. Nach Definition hätte jedes kürzere Stück einen nichtpositiven Score und würde uns nicht weiter helfen. Wir müssen dann nur noch prüfen, ob die obere Längenbeschränkung eingehalten wird. Dies ist in der folgenden Abbildung 1.31 veranschaulicht.

Mit Hilfe der linksnegativen Zeiger können wir das Problem mit dem in Abbildung 1.32 angegebenen Algorithmus leicht lösen. Wir versuchen hier ab jeder Position $i \in [1 : n]$ eine Maximal Scoring Subsequence der Länge höchstens λ zu finden.

Für die Laufzeit halten wir das Folgende fest. Nach maximal einer konstanten Anzahl von Operationen wird entweder i oder j erhöht, somit ist die Laufzeit $O(n)$.

Wir haben jedoch die linksnegativen Zeiger noch nicht berechnet. Wir durchlaufen dazu die Folge vom Ende zum Anfang hin. Treffen wir auf ein Element $a_i > 0$, dann sind wir fertig. Andernfalls verschmelzen wir das Element mit den folgenden Segmenten, bis das neu entstehende Segment einen positiven Score erhält. Dies ist

Abbildung 1.33: Skizze: Verschmelzen eines Elements mit Segmenten

in der folgenden Abbildung 1.33 illustriert, wobei nach dem Verschmelzen mit $A_j^{(i)}$ das Segment ab Position i einen positiven Score erhält.

Wir können diese Idee in den in Abbildung 1.34 angegebenen Algorithmus umsetzen, wobei $s(i) := \sigma(i, p(i))$ bezeichnet.

```
compute_left_negative_pointer (real a[], int n)
begin
    int i, p[n], s[n];
    for (i := n; i > 0; i−−) do
        p(i) := i;
        s(i) := a_i;
        while ((p(i) < n) && (s(i) ≤ 0)) do
            s(i) := s(i) + s(p(i) + 1);
            p(i) := p(p(i) + 1);
end
```

Abbildung 1.34: Algorithmus: Berechnung linksnegativer Zeiger durch Verschmelzen von Segmenten

Eine simple Laufzeit-Abschätzung ergibt, das die Laufzeit im schlimmsten Fall $O(n^2)$ beträgt. Mit einer geschickteren Analyse können wir jedoch wiederum eine lineare Laufzeit zeigen. Es wird pro Iteration maximal ein neues Segment generiert. In jeder inneren 'while-Schleife' wird ein Segment eliminiert. Es müssen mindestens so viele Segmente generiert wie gelöscht (= verschmolzen) werden. Da nur n Segmente generiert werden, können auch nur n Segmente gelöscht werden und somit ist die Laufzeit $O(n)$. Halten wir das Resultat im folgenden Satz fest.

Theorem 1.24 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine Folge reeller Zahlen und $\lambda \in \mathbb{N}$. Dann lässt sich eine Maximal Scoring Subsequence der Länge höchstens λ in Zeit $O(n)$ finden.*

Es gilt sogar der folgende Satz.

Theorem 1.25 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine Folge reeller Zahlen und $\underline{\lambda} \leq \overline{\lambda} \in \mathbb{N}$. Dann lässt sich eine Maximal Scoring Subsequence der Länge mindestens $\underline{\lambda}$ und höchstens $\overline{\lambda}$ in Zeit $O(n)$ finden.*

Beweis: Der Beweis sei dem Leser zur Übung überlassen. ■

1.5 Maximal Average Scoring Subsequence (*)

Jetzt wollen wir eine Teilfolge finden, deren Mittelwert (gemittelt über die Länge) maximal ist.

1.5.1 Problemstellung

Bevor wir zur Formalisierung des Problems kommen, führen wir zunächst noch einige abkürzende Notationen ein, die uns bei der Formulierung und dann auch bei der Lösung helfen werden.

Notation 1.26 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$, dann ist*

$$\sigma(i,j) := \sum_{k=i}^{j} a_k, \qquad \ell(i,j) := j - i + 1, \qquad \mu(i,j) := \frac{\sigma(i,j)}{\ell(i,j)}.$$

Damit lässt sich nun das Problem formalisieren.

Maximal Average Scoring Subsequence (MASS)

Eingabe: Eine Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$, $\lambda \in \mathbb{N}$.
Gesucht: Eine (zusammenhängende) Teilfolge $(a_i, \ldots, a_j)$ mit $\ell(i,j) \geq \lambda$, die $\mu(i,j)$ maximiert, d.h. $\mu(i,j) = \max\{\mu(i',j') \mid j' \geq i' + \lambda - 1\}$.

Wozu haben wir hier noch die untere Schranke λ eingeführt? Ansonsten wird das Problem trivial, denn dann ist das maximale Element, interpretiert als eine einelementige Folge, die gesuchte Lösung!

1.5.2 Rechtsschiefe Folgen und fallend rechtsschiefe Partitionen

Zur Lösung des Problems benötigen wir den Begriff einer rechtsschiefen Folge sowie einer fallend rechtsschiefen Partition.

Definition 1.27 *Eine Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ heißt* rechtsschief*, wenn für alle $i \in [1 : n-1]$ gilt: $\mu(1,i) \leq \mu(i+1,n)$.*

Eine Partition $a = A_1 \cdots A_k$ von a heißt fallend rechtsschief*, wenn jedes Segment A_i rechtsschief ist und $\mu(A_i) > \mu(A_j)$ für $i < j$ gilt.*

Anschaulich heißt eine Folge $a = (a_1, \ldots, a_n)$ also rechtsschief, wenn der Durchschnittswert jedes echten Präfix $(a_1, \ldots, a_i)$ kleiner gleich dem Durchschnittswert des korrespondierenden Suffixes $(a_{i+1}, \ldots, a_n)$ ist.

Lemma 1.28 *Seien $a \in \mathbb{R}^n$ und $b \in \mathbb{R}^m$ zwei reelle Folgen mit $\mu(a) < \mu(b)$. Dann gilt $\mu(a) < \mu(ab) < \mu(b)$ (ab sei die konkatenierte Folge).*

Beweis: Es gilt:

$$\begin{aligned}\mu(ab) &= \frac{\sigma(ab)}{\ell(ab)}\\ &= \frac{\sigma(a)+\sigma(b)}{\ell(ab)}\\ &= \frac{\mu(a)\cdot\ell(a)+\mu(b)\cdot\ell(b)}{\ell(ab)}.\end{aligned}$$

Somit ist zum einen (da $\mu(b) > \mu(a)$ ist)

$$\begin{aligned}\mu(ab) &= \frac{\mu(a)\cdot\ell(a)+\mu(b)\cdot\ell(b)}{\ell(ab)}\\ &> \frac{\mu(a)\cdot\ell(a)+\mu(a)\cdot\ell(b)}{\ell(ab)}\\ &= \frac{\mu(a)\cdot(\ell(a)+\ell(b))}{\ell(ab)}\\ &= \frac{\mu(a)\cdot\ell(ab)}{\ell(ab)}\\ &= \mu(a)\end{aligned}$$

und zum anderen (da $\mu(a) < \mu(b)$ ist)

$$\begin{aligned}\mu(ab) &= \frac{\mu(a)\cdot\ell(a)+\mu(b)\cdot\ell(b)}{\ell(ab)}\\ &< \frac{\mu(b)\cdot\ell(a)+\mu(b)\cdot\ell(b)}{\ell(ab)}\\ &= \frac{\mu(b)\cdot(\ell(a)+\ell(b))}{\ell(ab)}\\ &= \mu(b).\end{aligned}$$

Damit ist die Behauptung gezeigt. ■

Korollar 1.29 *Seien $a \in \mathbb{R}^n$ und $b \in \mathbb{R}^m$ zwei reelle Folgen mit $\mu(a) \leq \mu(b)$. Dann gilt $\mu(a) \leq \mu(ab) \leq \mu(b)$.*

Beweis: Gilt $\mu(a) < \mu(b)$, so ist es gerade die stärkere Aussage des vorherigen Lemmas. Gilt $\mu(a) = \mu(b)$, so gilt trivialerweise $\mu(a) = \mu(ab) = \mu(b)$. ■

Aus dem Beweis des vorherigen Lemmas folgt sofort auch das folgende Korollar.

Korollar 1.30 *Seien $a \in \mathbb{R}^n$ und $b \in \mathbb{R}^m$ zwei reelle Folgen mit $\mu(a) > \mu(b)$. Dann gilt $\mu(a) > \mu(ab) > \mu(b)$.*

Lemma 1.31 *Seien $a \in \mathbb{R}^n$ und $b \in \mathbb{R}^m$ zwei rechtsschiefe Folgen mit $\mu(a) \leq \mu(b)$. Dann ist auch die Folge ab rechtsschief.*

Beweis: Sei p ein beliebiges Präfix von ab. Es ist zu zeigen, dass $\mu(p) \leq \mu(q)$ ist, wobei q das zu p korrespondierende Suffix in ab ist, d.h. $ab = pq$.

Fall 1: Sei $p = a$. Dann gilt $\mu(a) \leq \mu(b)$ nach Voraussetzung, und die Behauptung ist erfüllt, da $q = b$ ist.

Fall 2: Sei jetzt p ein echtes Präfix von a, d.h. $a = pa'$. Da a eine rechtsschiefe Folge ist, gilt $\mu(p) \leq \mu(a')$. Mit dem vorherigen Korollar gilt dann:

$$\mu(p) \leq \mu(pa') \leq \mu(a').$$

Somit ist $\mu(p) \leq \mu(a) \leq \mu(b)$.

Mit $\mu(p) \leq \mu(a')$ und $\mu(p) \leq \mu(b)$, folgt

$$\begin{aligned} \mu(p) &= \frac{\ell(a')\mu(p) + \ell(b)\mu(p)}{\ell(a'b)} \\ &\leq \frac{\ell(a')\mu(a') + \ell(b)\mu(b)}{\ell(a'b)} \\ &= \frac{\sigma(a') + \sigma(b)}{\ell(a'b)} \\ &= \mu(a'b) \\ &= \mu(q). \end{aligned}$$

Fall 3: Sei $p = ab'$ mit $b = b'b''$. Mit dem vorherigen Korollar folgt, da $b = b'b''$ rechtsschief ist:

$$\mu(b') \leq \mu(\underbrace{b'b''}_{=b}) \leq \mu(b'').$$

Somit gilt $\mu(a) \leq \mu(b) \leq \mu(b'')$. Also gilt mit $\mu(a) \leq \mu(b'')$ und $\mu(b') \leq \mu(b'')$:

$$\begin{aligned}
\mu(p) &= \mu(ab') \\
&= \frac{\sigma(a) + \sigma(b')}{\ell(ab')} \\
&= \frac{\ell(a)\mu(a) + \ell(b')\mu(b')}{\ell(ab')} \\
&\leq \frac{\ell(a)\mu(b'') + \ell(b')\mu(b'')}{\ell(ab')} \\
&= \mu(b'') \\
&= \mu(q).
\end{aligned}$$

Damit ist das Lemma bewiesen. ■

Lemma 1.32 *Jede Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ besitzt eine eindeutige fallend rechtsschiefe Partition.*

Beweis: Wir führen den Beweis durch Induktion über n.

Induktionsanfang ($n = 1$): Klar, nach Definition.

Induktionsschritt ($n \to n + 1$): Wir betrachten eine Folge $(a_1, \ldots, a_n, a_{n+1})$. Sei weiter $A_1 \cdots A_k$ die fallend rechtsschiefe Partition von $(a_1, \ldots, a_n)$.

Gilt $\mu(A_k) > a_{n+1} = \mu(a_{n+1})$, dann ist $A_1 \cdots A_k \cdot (a_{n+1})$ eine neue rechtsschiefe Partition.

Andernfalls bestimmen wir ein maximales i mit $\mu(A_{i-1}) > \mu(A_i \cdots A_k \cdot (a_{n+1}))$. Dann behaupten wir, dass die neue Partition $A_1 \cdots A_{i-1} \cdot (A_i \cdots A_k \cdot (a_{n+1}))$ eine fallend rechtsschiefe ist.

Nach Definition ist (a_{n+1}) rechtsschief. Nach dem vorherigen Lemma 1.31 und der Wahl von i ist dann auch $A_k \cdot (a_{n+1})$ rechtsschief. Weiter gilt allgemein für $j \geq i$, dass $A_j \cdots A_k \cdot (a_{n+1})$ rechtsschief ist. Somit ist auch $A_i \cdots A_k \cdot (a_{n+1})$ rechtsschief.

Nach Konstruktion ist also die jeweils konstruierte Partition eine fallend rechtsschiefe Partition.

Es bleibt noch die Eindeutigkeit zu zeigen. Nehmen wir an, es gäbe zwei verschiedene fallend rechtsschiefe Partitionen. Betrachten wir, wie in der folgenden Abbildung 1.35 skizziert, die jeweils linkesten Teilfolgen in ihren Partition, in denen sich die beiden Partitionen unterscheiden.

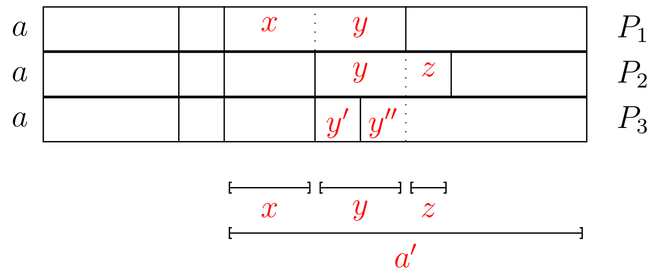


Abbildung 1.35: Skizze: Beweis der Eindeutigkeit der rechtsschiefen Partition

Betrachten wir zuerst den Fall P_1 gegen P_2. Sei zuerst $z = \varepsilon$. Dann gilt $\mu(x) > \mu(y)$, da P_2 eine fallend rechtsschiefe Partition ist. Da xy in P_1 rechtsschief ist, gilt $\mu(x) \leq \mu(y)$ und wir erhalten den gewünschten Widerspruch.

Sei nun $z \neq \varepsilon$. Da yz nach der Partition P_2 rechtsschief ist, gilt $\mu(y) \leq \mu(z)$. Mit Korollar 1.29 folgt, dass $\mu(y) \leq \mu(yz) \leq \mu(z)$.

Nach Wahl der fallend rechtsschiefen Partition P_2 gilt $\mu(x) > \mu(yz)$. Wie wir eben gezeigt haben, gilt auch $\mu(yz) \geq \mu(y)$. Damit ist $\mu(x) > \mu(y)$ und somit ist xy nicht rechtsschief. Dies ist ein Widerspruch zur Annahme, dass P_1 eine fallend rechtsschiefe Partition ist.

Es bleibt noch der Fall P_1 gegen P_3 zu betrachten. Nach P_1 gilt $\mu(x) \leq \mu(y)$ und nach P_3 gilt $\mu(x) > \mu(y')$. Also gilt $\mu(y) > \mu(y')$. Sei y'' so gewählt, dass $y = y'y''$ und sei weiter $y = y' \cdot Y_2 \cdots Y_k$ eine fallende rechtsschiefe Partition von y, die sich aus P_3 bzw. dem ersten Teil des Beweises ergibt. Dann gilt $\mu(y') > \mu(Y_2) > \cdots > \mu(Y_k)$. Dann muss aber $\mu(y') > \mu(y'')$ sein. Somit gilt $\mu(y) > \mu(y') > \mu(y'')$, was ein offensichtlicher Widerspruch ist. ∎

1.5.3 Algorithmus zur Konstruktion rechtsschiefer Zeiger

Kommen wir nun zu einem Algorithmus, der die fallend rechtsschiefe Partition zu einer gegebenen Folge konstruiert. Zuerst benötigen wir noch einige Notationen.

Notation 1.33 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Jedes Suffix $a^{(i)} = (a_i, \ldots, a_n)$ von a besitzt eine eindeutige fallend rechtsschiefe Partition: $a^{(i)} = A_1^{(i)} \cdots A_{k_i}^{(i)}$.*

Damit können wir die so genannten rechtsschiefen Zeiger definieren.

Definition 1.34 *Sei* $A_1^{(i)} = (a_i, \ldots, a_{p(i)})$*, dann heißt* $p(i)$ *der* rechtsschiefe Zeiger *von* i*.*

Für den folgenden Algorithmus benötigen wir noch die folgenden vereinfachenden Notationen.

Notation 1.35 *Im Folgenden verwenden wir der Einfachheit halber die beiden folgenden Abkürzungen:*

$$\begin{aligned} s(i) &:= \sigma(i, p(i)), \\ \ell(i) &:= \ell(i, p(i)) = p(i) - i + 1. \end{aligned}$$

Für die Konstruktion der rechtsschiefen Zeiger arbeiten wir uns wieder vom Ende zum Anfang durch die gegebene Folge durch. Für ein neu betrachtetes Element setzen wir zunächst die rechtsschiefe Folge auf diese einelementige Folge. Ist nun der

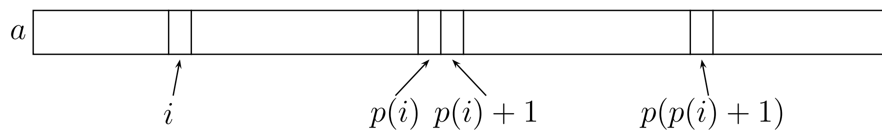


Abbildung 1.36: Skizze: Verschmelzen von Segmenten

Mittelwert des aktuell betrachten Segments kleiner gleich dem Mittelwert des folgenden Segments, so verschmelzen wir diese beiden und betrachten dieses neue Segment als das aktuelle. Andernfalls haben wir die Eigenschaft einer fallend rechtsschiefen Partition sichergestellt. Dies ist in der Abbildung 1.36 schematisch dargestellt.

Da diese eindeutig ist, erhalten wir zum Schluss die gewünschte fallend rechtsschiefe Partition samt aller rechtsschiefer Zeiger. Somit können wir den in Abbildung 1.37 angegebenen Algorithmus zur Konstruktion rechtsschiefer Zeiger formalisieren.

Lemma 1.36 *Die rechtsschiefen Zeiger lassen sich in Zeit* $O(n)$ *berechnen.*

Beweis: Analog zum Beweis der Laufzeit des Algorithmus zur Konstruktion linksnegativer Zeiger. ∎

```
compute_rightskew_pointer (real a[], int n)
begin
    for (i := n; i > 0; i--) do
        p(i) := i;
        s(i) := a_i;
        ℓ(i) := 1;
        while ((p(i) < n) && (s(i)/ℓ(i) ≤ s(p(i) + 1)/ℓ(p(i) + 1))) do
            s(i) := s(i) + s(p(i) + 1);
            ℓ(i) := ℓ(i) + ℓ(p(i) + 1);
            p(i) := p(p(i) + 1);
end
```

Abbildung 1.37: Algorithmus: Rechtsschiefe Zeiger

1.5.4 Elementare Eigenschaften von MASS

Bevor wir zur Bestimmung von Teilfolgen vorgegebener Mindestlänge mit einem maximalem Mittelwert kommen, werden wir erst noch ein paar fundamentale Eigenschaften festhalten.

Lemma 1.37 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ und $(a_i, \ldots, a_j)$ eine kürzeste Teilfolge von a der Länge mindestens λ, die deren Average Score $\mu(i, j)$ maximiert. Dann gilt $\ell(i, j) = j - i + 1 \leq 2\lambda - 1$.*

Beweis: Der Beweis sei dem Leser zur Übung überlassen. ■

Lemma 1.38 *Seien $a \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ und $c \in \mathbb{R}^k$ drei reelle Folgen, die die Beziehung $\mu(a) < \mu(b) < \mu(c)$ erfüllen. Dann gilt $\mu(ab) < \mu(abc)$.*

Zunächst geben wir ein Gegenbeispiel an, um zu zeigen, dass im vorherigen Lemma die Bedingung $\mu(a) < \mu(b)$ notwendig ist. Sei $a = 11$, $b = 1$ und $c = 3$, dann ist $\mu(ab) = \frac{12}{2} = 6$ sowie $\mu(abc) = \frac{15}{3} = 5$.

Beweis: Nach Korollar 1.30 gilt $\mu(a) < \mu(ab) < \mu(b) < \mu(c)$. Somit gilt insbesondere $\mu(ab) < \mu(c)$. Nach Korollar 1.30 gilt dann $\mu(ab) < \mu(abc) < \mu(c)$ und das Lemma ist bewiesen. ■

Lemma 1.39 *Seien $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ und $p \in \mathbb{R}^\ell$ und sei $A_1 \cdots A_k$ die fallend rechtsschiefe Partition von a. Sei weiter m maximal gewählt, so dass*

$$\mu(p \cdot A_1 \cdots A_m) = \max\{\mu(p \cdot A_1 \cdots A_i) \mid i \in [0 : k]\}.$$

Es gilt genau dann $\mu(p \cdot A_1 \cdots A_i) > \mu(A_{i+1})$, wenn $i \geq m$ gilt.

Beweis: $\Rightarrow$: Sei i so gewählt, dass $\mu(p \cdot A_1 \cdots A_i) > \mu(A_{i+1})$ gilt. Da $A_1 \cdots A_k$ die fallend rechtsschiefe Partition von a ist, gilt $\mu(A_1) > \mu(A_2) > \cdots > \mu(A_k)$. Dann gilt auch $\mu(p \cdot A_1 \cdots A_i) > \mu(A_{i+1}) > \mu(A_{i+2}) > \cdots > \mu(A_k)$. Nach Korollar 1.30 gilt dann $\mu(p \cdot A_1 \cdots A_i) > \mu(p \cdot A_1 \cdots A_{i+1}) > \mu(A_{i+1}) > \cdots > \mu(A_k)$. Mit Korollar 1.30 gilt also auch $\mu(p \cdot A_1 \cdots A_i) > \mu(p \cdot A_1 \cdots A_j) > \mu(A_j) > \cdots > \mu(A_k)$ für alle $j \in [i : k]$. Aufgrund der Definition von m muss also $i \geq m$ gelten.

$\Leftarrow$: Zum Beweis der Behauptung unterscheiden wir die folgenden beiden Fälle, je nachdem, ob $\mu(p \cdot A_1 \cdots A_m) > \mu(A_{m+1})$ oder ob $\mu(p \cdot A_1 \cdots A_m) \leq \mu(A_{m+1})$ gilt.

Fall 1: Es gilt $\mu(p \cdot A_1 \cdots A_m) > \mu(A_{m+1}) > \mu(A_{m+2}) > \cdots > \mu(A_k)$. Korollar 1.30 liefert $\mu(p \cdot A_1 \cdots A_{m+1}) > \mu(A_{m+1}) > \mu(A_{m+2}) > \cdots > \mu(A_k)$. Nach wiederholter Anwendung von Korollar 1.30 gilt dann für $j \in [1 : k - m]$ natürlich auch $\mu(p \cdot A_1 \cdots A_{m+j}) > \mu(A_{m+j}) > \mu(A_{m+j+1}) > \cdots > \mu(A_k)$. Also gilt für $i \geq m$ $\mu(p \cdot A_1 \cdots A_i) > \mu(A_i) > \mu(A_{i+1}) > \cdots > \mu(A_k)$ und somit die Behauptung.

Fall 2: Es gilt $\mu(p \cdot A_1 \cdots A_m) \leq \mu(A_{m+1})$. Nach Korollar 1.29 gilt nun allerdings $\mu(p \cdot A_1 \cdots A_m) \leq \mu(p \cdot A_1 \cdots A_{m+1}) \leq \mu(A_{m+1})$. Aufgrund der Wahl von m (das Maximum von $\mu(p \cdot A_1 \cdots A_i)$ wird für $i = m$ angenommen) gilt dann $\mu(p \cdot A_1 \cdots A_m) = \mu(p \cdot A_1 \cdots A_{m+1})$. Daraus ergibt sich ein Widerspruch zur Maximalität von m. Damit ist die Behauptung des Lemmas gezeigt. ■

1.5.5 Ein Algorithmus für MASS

Somit können wir einen Algorithmus zur Lösung unseres Problems angeben, der im Wesentlichen auf Lemma 1.39 basiert. Wir laufen von links nach rechts durch die Folge, und versuchen, ab Position i die optimale Sequenz mit maximalem Mittelwert zu finden. Nach Voraussetzung hat diese mindestens die Länge λ und nach Lemma 1.37 auch höchstens die Länge $2\lambda - 1$. Dies ist in Abbildung 1.38 illustriert.

Ein einfacher Algorithmus würde jetzt alle möglichen Längen aus dem Intervall $[\lambda : 2\lambda - 1]$ ausprobieren. Der folgende, in Abbildung 1.39 angegebene Algorithmus würde dies tun, wenn die Prozedur `locate` geeignet implementiert wäre, wobei `locate` einfach die Länge einer optimalen Teilfolge zurückgibt. Würden wir `locate`

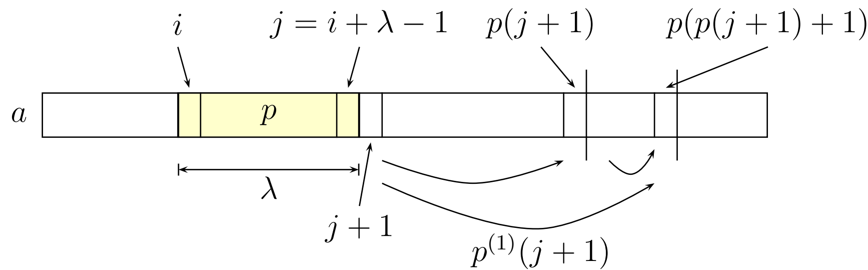


Abbildung 1.38: Skizze: Auffinden der optimalen Teilfolge

mittels einer linearen Suche implementieren, so würden wir einen Laufzeit von $O(n\lambda) = O(n^2)$ erhalten.

Mit Hilfe von Lemma 1.39 können wir aber auch geschickter vorgehen. Das Lemma besagt nämlich, dass wenn wir rechts vom optimalen Schnitt sind, der Mittelwert größer als der des folgenden Segments ist. Links vom Optimum gilt, dass der Wert kleiner als der Mittelwert vom folgenden Segment ist. Somit könnten wir hier eine binäre Suche ausnutzen.

Allerdings kennen wir die Grenzen der Segmente nicht explizit, sondern nur als lineare Liste über die rechtsschiefen Zeiger. Daher konstruieren wir uns zu den rechtsschiefen Zeiger noch solche, die nicht nur auf das nächste Segment, sondern auch auf das 2^k-te folgende Segment angibt. Dafür definieren wir erst einmal formal die Anfänge der nächsten 2^k-ten Segment sowie die darauf basierenden iterierten

```
MASS (real a[], int n, λ)
begin
    int mi := 1;
    int mj := λ;
    int mm =: μ(mi, mj);
    for (i := 1; i ≤ n; i++) do
        j := i + λ − 1;
        if (μ(i, j) ≤ μ(j + 1, p(j + 1))) then
            j := locate(i, j);
        if (μ(i, j) > mm) then
            mi := i;
            mj := j;
            mm := μ(mi, mj);
end
```

Abbildung 1.39: Algorithmus: MASS

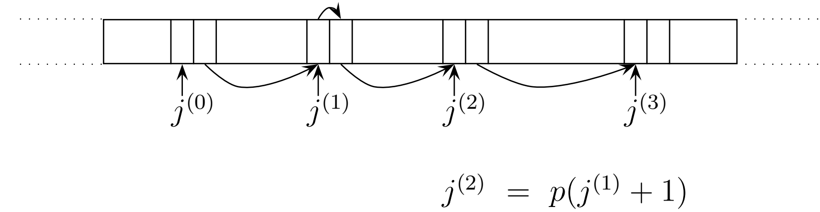


Abbildung 1.40: Skizze: Definition der Werte $j^{(\cdot)}$

rechtsschiefen Zeiger wie folgt:

$$\begin{aligned} j^{(0)} &:= j, \\ j^{(k)} &= \min\{p(j^{(k-1)} + 1), n\}, \\ p^{(0)}(i) &= p(i), \\ p^{(k)}(i) &= \min\{p^{(k-1)}(p^{(k-1)}(i) + 1), n\}. \end{aligned}$$

Hierbei gibt $p^{(k)}(i)$ das Ende nach 2^k Segmenten an. Dies ist in folgenden Abbildungen illustriert, in Abbildung 1.40 die Definition von $j^{(\cdot)}$ und in Abbildung 1.41 die Definition von $p^{(\cdot)}$.

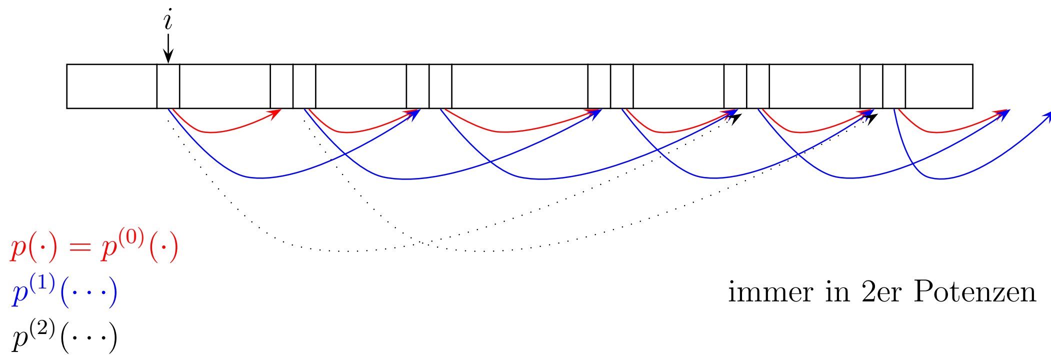


Abbildung 1.41: Skizze: Definition der Werte $p^{(\cdot)}$

Definition 1.40 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge und $p(\cdot)$ die zugehörigen rechtsschiefen Zeiger. Der k-te* iterierte rechtsschiefe Zeiger *$p^{(k)}(i)$ ist rekursiv definiert durch $p^{(k)}(i) := \min\{p^{(k-1)}(p^{(k-1)}(i) + 1), n\}$ und $p^{(0)}(i) := p(i)$.*

Diese iterierten rechtsschiefen Zeiger lassen sich aus den rechtsschiefen Zeigern in Zeit $O(\log(\lambda))$ berechnen, da wir ja maximal die 2λ-iterierten rechtsschiefen Zeiger aufgrund der oberen Längenbeschränkung von $2\lambda - 1$ benötigen.

Mit Hilfe dieser iterierten rechtsschiefen Zeiger können wir jetzt die binäre Suche in der Prozedur `locate` wie im folgenden Algorithmus implementieren, der in Abbildung 1.42 angegeben ist. Hierbei ist zu beachten, dass die erste zu testende Position bei der binären Suche bei maximal $\log(2\lambda - 1) - 1 \leq \log(\lambda)$ liegt.

```
locate (int i, j)
begin
    for (k := log(λ); k ≥ 0; k−−) do
        if  ((j ≥ n) || (μ(i, j) > μ(j + 1, p(j + 1)))) then
            return j;
        if ((p^(k)(j + 1) < n) &&
        (μ(i, p^(k)(j + 1)) ≤ μ(p^(k)(j + 1) + 1, p(p^(k)(j + 1) + 1)))) then
            j := p^(k)(j + 1);
    if ((j < n) && (μ(i, j) ≤ μ(j + 1, p(j + 1)))) then
        j := p(j + 1);
    return j;
end
```

Abbildung 1.42: Algorithmus: locate(i, j)

Die Strategie der Suche ist noch einmal in der folgenden Abbildung 1.43 illustriert.

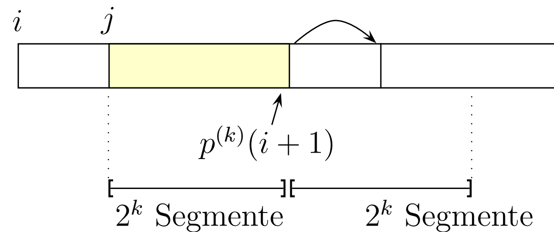


Abbildung 1.43: Skizze: Strategie von `locate`

Zusammenfassend erhalten wir das folgende Theorem.

Theorem 1.41 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Eine kürzeste Teilfolge der Länge mindestens λ mit maximalem durchschnittlichem Score kann in Zeit $O(n \log(\lambda))$ gefunden werden.*

Wir wollen hier noch erwähnen, dass mittlerweile für das MASS-Problem Algorithmen mit linearer Laufzeit bekannt sind.

1.6 Weighted Maximal Average Scoring Subsequence (*)

In diesem Abschnitt skizzieren wir einen linearen Algorithmus für die Bestimmung einer optimalen Teilfolge bezüglich des Mittelwerts. Darüber hinaus ist dies sogar auch für einen obere Längenbeschränkung möglich.

1.6.1 Problemstellung

Das Problem lässt sich nun wie folgt in etwas allgemeinerer Fassung formalisieren.

WEIGHTED MAXIMAL AVERAGE SCORING SUBSEQUENCE (WMASS)

Eingabe: Eine Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ mit Gewichten $w = (w_1, \ldots, w_n) \in \mathbb{N}^n$ und $\lambda \in \mathbb{N}$.

Gesucht: Eine (zusammenhängende) Teilfolge $(a_i, \ldots, a_j)$ mit $\ell(i, j) \geq \lambda$, die den Wert $\mu(i, j) = \sigma(i, j)/\ell(i, j)$ maximiert, wobei $\sigma(i, j) = \sum_{k=1}^{j} a_k$ und $\ell(i, j) = \sum_{k=i}^{j} w_k$.

Setzt man $w_k = 1$ für alle $k \in [1 : n]$ so erhält man die Definition aus dem vorherigen Abschnitt bzw. das entsprechende Analogon der ersten Abschnitte. Die Gewichte müssen keine natürlichen Zahlen sein, man kann auch positive reelle Zahlen verwenden. Auch die Einführung einer oberen Schranke ist möglich, aber der Kürze wegen wollen hier nicht näher darauf eingehen, sondern verweisen auf die Literatur von Chung und Lu.

Wozu haben wir hier noch die untere Schranke λ eingeführt? Ansonsten wird das Problem trivial, denn für $w_i = 1$ ist dann das maximale Element, interpretiert als eine einelementige Folge, die gesuchte Lösung! Für beliebige Gewichte ist dann das Element mit dem maximalen relativen Verhältnis a_i/w_i die Lösung!

Bevor wir zur Bestimmung von Teilfolgen vorgegebener Mindestlänge mit einem maximalem Mittelwert kommen, werden wir erst noch eine fundamentale Eigenschaft festhalten.

Lemma 1.42 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ mit $w = (1, \ldots, 1)$ und $(a_i, \ldots, a_j)$ eine kürzeste Teilfolge von a der Länge mindestens λ, die deren Average Score $\mu(i, j)$ maximiert. Dann gilt $\ell(i, j) = j - i + 1 \leq 2\lambda - 1$.*

Beweis: Der Beweis sei dem Leser zur Übung überlassen. ■

1.6.2 Elementare Eigenschaften

Zuerst halten wir einige fundamentale Eigenschaften über den gewichteten Durchschnitt von Teilfolgen fest.

Lemma 1.43 *Sei $(a_1, \ldots, a_n) \in \mathbb{R}^n$ eine Folge mit Gewichten $(w_1, \ldots, w_n) \in \mathbb{N}^n$. Dann gilt für alle $x \leq y < z \in [1 : n]$:*

i) $\mu(x, y) < \mu(y + 1, z) \quad \Leftrightarrow \quad \mu(x, z) < \mu(y + 1, z) \quad \Leftrightarrow \quad \mu(x, y) < \mu(x, z)$,

ii) $\mu(x, y) > \mu(y + 1, z) \quad \Leftrightarrow \quad \mu(x, z) > \mu(y + 1, z) \quad \Leftrightarrow \quad \mu(x, y) > \mu(x, z)$.

Beweis: Es gelte $\mu(x, y) < \mu(y + 1, z)$, d.h. $\sigma(x, y) < \frac{\ell(x,y)}{\ell(y+1,z)}\sigma(y + 1, z)$, dann gilt:

$$\begin{aligned}
\mu(x, z) &= \frac{\sigma(x, y) + \sigma(y + 1, z)}{\ell(x, z)} \\
&< \frac{\ell(x, y)\sigma(y + 1, z)}{\ell(x, z)\ell(y + 1, z)} + \frac{\sigma(y + 1, z)}{\ell(x, z)} \\
&= \frac{\sigma(y + 1, z)}{\ell(x, z)} \left(\frac{\ell(x, y)}{\ell(y + 1, z)} + 1 \right) \\
&= \frac{\sigma(y + 1, z)}{\ell(x, z)} \left(\frac{\ell(x, y) + \ell(y + 1, z)}{\ell(y + 1, z)} \right) \\
&= \frac{\sigma(y + 1, z)}{\ell(y + 1, z)} \\
&= \mu(y + 1, z).
\end{aligned}$$

Es gelte $\mu(x, z) < \mu(y + 1, z)$, d.h. $\sigma(y + 1, z) > \frac{\ell(y+1,z)}{\ell(x,z)}\sigma(x, z)$, dann gilt:

$$\begin{aligned}
\mu(x, y) &= \frac{\sigma(x, z) - \sigma(y + 1, z)}{\ell(x, y)} \\
&< \frac{1}{\ell(x, y)} \left(\sigma(x, z) - \frac{\ell(y + 1, z)\sigma(x, z)}{\ell(x, z)} \right) \\
&= \frac{\sigma(x, z)}{\ell(x, z)} \left(\frac{\ell(x, z)}{\ell(x, y)} - \frac{\ell(y + 1, z)}{\ell(x, y)} \right) \\
&= \frac{\sigma(x, z)}{\ell(x, z)} \\
&= \mu(x, z).
\end{aligned}$$

Es gelte $\mu(x,y) < \mu(x,z)$, d.h. $\sigma(x,z) > \ell(x,z)\frac{\sigma(x,y)}{\ell(x,y)}$, dann gilt:

$$\begin{aligned}
\mu(x,y) &= \frac{\sigma(x,y)}{\ell(x,y)}\left(\frac{\ell(x,z)-\ell(x,y)}{\ell(y+1,z)}\right) \\
&= \frac{1}{\ell(y+1,x)}\left(\frac{\ell(x,z)\sigma(x,y)}{\ell(x,y)} - \frac{\ell(x,y)\sigma(x,y)}{\ell(x,y)}\right) \\
&< \frac{1}{\ell(y+1,x)}\left(\sigma(x,z)-\sigma(x,y)\right) \\
&= \frac{\sigma(y+1,z)}{\ell(y+1,x)} \\
&= \mu(y+1,z).
\end{aligned}$$

Teil ii) wird völlig analog bewiesen. ■

Das folgende Korollar ergibt sich unmittelbar aus dem vorhergenden Lemma.

Korollar 1.44 *Sei $(a_1,\ldots,a_n) \in \mathbb{R}^n$ eine Folge mit Gewichten $(w_1,\ldots,w_n) \in \mathbb{N}^n$. Dann gilt für alle $x < y < z \in [1:n]$:*

i) $\mu(x,y) \leq \mu(y+1,z) \quad \Leftrightarrow \quad \mu(x,z) \leq \mu(y+1,z) \quad \Leftrightarrow \quad \mu(x,y) \leq \mu(x,z)$,

ii) $\mu(x,y) \geq \mu(y+1,z) \quad \Leftrightarrow \quad \mu(x,z) \geq \mu(y+1,z) \quad \Leftrightarrow \quad \mu(x,y) \geq \mu(x,z)$.

1.6.3 Generischer Algorithmus und seine Korrektheit

In diesem Abschnitt werden wir einen generischen Algorithmus für das Problem vorstellen und seine Korrektheit beweisen. Für die Beschreibung des Algorithmus benötigen wir noch die folgenden Notationen.

Notation 1.45 *Sei $(a_1,\ldots,a_n) \in \mathbb{R}^n$ eine Folge mit Gewichten $(w_1,\ldots,w_n) \in \mathbb{N}^n$ und $\lambda \in \mathbb{N}$, dann ist:*

- $\varphi(x,y) = \max\{z \in [x:y] \,:\, \mu(x,z) = \min\{\mu(x,z') \,:\, z' \in [x:y]\}\}$;
- $j_0 = \min\{j \in \mathbb{N} \,:\, \ell(1,j) \geq \lambda\}$;
- $r_j = \max\{i \in \mathbb{N} \,:\, \ell(i,j) \geq \lambda\}$;
- $i_j^* = \max\{k \in [1:r_j] \,:\, \mu(k,j) = \max\{\mu(i,j) \,:\, i \in [1:r_j]\}\}$;
- $j^* = \min\left\{j \in [1:n] \,:\, \mu(i_j^*,j) = \max\{\mu(i,j) \,:\, j \in [j_0:n] \wedge i \in [1:r_j]\}\right\}$.

Die Funktion $\varphi(x, y)$ liefert die Endposition eines längsten Präfixes von $(a_x, \ldots, a_y)$, die einen minimalen Average Score unter allen Präfixen besitzt. j_0 beschreibt die erste Endposition, an der eine Sequenz mit maximalen Average Score enden kann. Weiter beschreibt r_j die größte Indexposition i, die für eine Betrachtung der Teilfolge (i, j) möglich ist, andernfalls wird die untere Schranke λ verletzt. Der Wert i_j^* beschreibt die letztmögliche Startposition einer Teilfolge, die an Position j endet und den Average Score maximiert. Letztendlich ist j^* die Endposition der ersten Teilfolge, die den Average Score maximiert.

```
WMASS (real a[], real w[], int n, int λ)
begin
    int opt := 0;
    int L := 0;
    int R := 0;
    int i_{j0-1} := 1;
    for (j := j0; j ≤ n; j++) do
        i_j := Best(i_{j-1}, r_j, j);
        if (μ(i_j, j) > opt) then
            opt =: μ(i_j, j);
            L := i_j;
            R := j;
end

Best(ℓ, r, j)
begin
    int i := ℓ;
    while ((i < r) && (μ(i, φ(i, r − 1)) ≤ μ(i, j))) do
        i := φ(i, r − 1) + 1;
    return i;
end
```

Abbildung 1.44: Algorithmus: Generischer Algorithmus für WMASS

In dem in Abbildung 1.44 angegebenen Algorithmus bestimmen wir für jede mögliche Endposition j den zugehörigen Startwert i_j, der den Average Score maximiert. Dies wird mithilfe der Prozedur Best erreicht. Im Folgenden versuchen wir zuerst die Korrektheit zu beweisen. Im nächsten Abschnitt werden wir dann eine effiziente Implementierung vorstellen (hier ist ja die Realisierung der Funktion φ noch völlig offen). Wir beweisen zunächst eine wichtige Eigenschaft der Prozedur Best.

Lemma 1.46 *Für $\ell \leq r \leq j$ liefert* Best(ℓ, r, j) *den größten Index $i \in [\ell : r]$, der $\mu(i, j)$ maximiert.*

Beweis: Sei $i^* = \operatorname{argmax}\{\mu(i,j) : i \in [\ell : r]\}$. Ist das Maximum nicht eindeutig, so wählen wir den größten Index. Weiter sei $i_j := \text{Best}(\ell, r, j)$.

Wir führen den Beweis durch Widerspruch, also gilt $i_j \neq i^*$. O.B.d.A nehmen wir an, dass j der kleinste Index ist, für den Best nicht korrekt ist.

Fall 1 ($i_j < i^*$): Somit gilt auch $i_j < r$, da $i^* \leq r$ gelten muss. Aufgrund der while-Schleife im Algorithmus 1.44 gilt, dass $\mu(i_j, \varphi(i_j, r-1)) > \mu(i_j, j)$.

Da nach Definition von i^* gilt, dass $\mu(i_j, j) \leq \mu(i^*, j)$, folgt mit Korollar 1.44: $\mu(i_j, i^*-1) \leq \mu(i_j, j)$. Somit gilt insgesamt:

$$\mu(i_j, i^*-1) \leq \mu(i_j, j) < \mu(i_j, \varphi(i_j, r-1)).$$

Dies ist offensichtlich ein Widerspruch zur Definition von φ.

Fall 2 ($i_j > i^*$): Somit gilt $\ell < r$, da sonst $\ell = r$ und damit $i^* = r = i_j$.

Da $\ell < r$ und $i_j > i^* \geq \ell$ gilt, wird die while-Schleife im Algorithmus 1.44 mindestens einmal durchlaufen (da er ja $i_j > i^* \geq \ell$ zurückliefert). Daher gilt:

$$\exists i \in [\ell : r] : i < r \wedge \mu(i, \varphi(i, r-1)) \leq \mu(i, j) \wedge i \leq i^* < \varphi(i, r-1) + 1.$$

Wenn $i = i^*$, folgt mit Korollar 1.44:

$$\mu(i^*, \varphi(i^*, r-1)) \leq \mu(i^*, j) \leq \mu(\varphi(i^*, r-1) + 1, j).$$

Dies ist aber ein Widerspruch zur Definition von i^*, da $\varphi(i^*, r-1) + 1 > i^*$.

Also ist im Folgenden $i < i^*$ und es gilt (aufgrund der Definition von i^*):

$$\begin{aligned} \mu(i^*, j) &\geq \mu(i, j) \\ &\quad \text{Korollar 1.44 auf letzte Ungleichung} \\ &\geq \mu(i, i^*-1) \\ &\quad \text{da } i^*-1 \in [i : r-1] \text{ und Defintion von } \varphi \\ &\geq \mu(i, \varphi(i, r-1)) \\ &\quad \text{da } i^*-1 < \varphi(i, r-1) \\ &\quad \text{liefert das Korollar 1.44 auf die letzte Ungleichung} \\ &\geq \mu(i^*, \varphi(i, r-1)) \end{aligned}$$

Somit gilt also $\mu(i^*, j) \geq \mu(i^*, \varphi(i, r-1))$. Korollar 1.44 angewendet auf diese Ungleichung liefert

$$\mu(\varphi(i, r-1) + 1, j) \geq \mu(i^*, j).$$

Da $i^* < \varphi(i, r-1) + 1$ gilt, ist das ein Widerspruch zur Definition von i^*. ■

Basierend auf dem letzten Lemma zeigen wir jetzt die Korrektheit des generischen Algorithmus.

Theorem 1.47 *Algorithmus* WMASS *ist korrekt.*

Beweis: Da der Algorithmus WMASS alle möglichen $j \in [j_0 : n]$ durchprobiert und für $j = j^*$ als Ergebnis i_{j^*} liefert, genügt es zu zeigen, dass $i_{j^*} = i^*_{j^*}$.

Fall 1 ($j^* = j_0$): Da $i_{j_0-1} = 1$, gilt aufgrund des Algorithmus $i_{j_0} = \textsc{Best}(1, r_j, j_0)$. Da $j_0 = j^* \in [1 : r_j]$ sein muss, folgt aus Lemma 1.46 die Korrektheit.

Fall 2 ($j^* > j_0$): Irgendwann erfolgt im Laufe des Algorithmus WMASS der Aufruf $\textsc{Best}(i_{j*-1}, r_{j^*}, j^*)$. Wegen Lemma 1.46 genügt es also zu zeigen, dass $i^*_{j^*-1} \leq i^*_{j^*}$.

Dies zeigen wir durch einen Widerspruchsbeweis. Sei im Folgenden für $j \in [j_0 : n]$ wieder $i_j := \textsc{Best}(i_{j-1}, r_j, j)$. Wir nehmen also an, dass ein $j \in [j_0 : j^* - 1]$ existiert mit $i_{j-1} \leq i^*_{j^*} < i_j$.

Zunächst gilt aufgrund der Definition von j^* und $i^*_{j^*} < i_j$, dass $\mu(i^*_{j^*}, j^*) > \mu(i_j, j^*)$. Mit dem Lemma 1.43 folgt dann

$$\mu(i^*_{j^*}, i_j - 1) > \mu(i^*_{j^*}, j^*). \tag{1.1}$$

Mit $i_{j-1} \leq i^*_{j^*} < i_j \leq r_j$ und dem Lemma 1.46 folgt:

$$\begin{aligned}
\mu(i_j, j) &\geq \mu(i^*_{j^*}, j) \\
&\ \text{mit Korollar 1.44 auf die letzte Ungleichung} \\
&\geq \mu(i^*_{j^*}, i_j - 1) \\
&\ \text{wegen der Ungleichung (1.1)} \\
&> \mu(i^*_{j^*}, j^*).
\end{aligned}$$

Das ist aber ein Widerspruch zur Definition von j^*. ■

Wir wollen noch kurz die Laufzeit des generischen Algorithmus analysieren, vorausgesetzt, dass die Funktion φ in konstanter Zeit bestimmt werden kann. In der Schleife wird für j das Intervall $[i_{j-1} : i_j]$ durchlaufen. Für $j + 1$ anschließend das Intervall $[i_j : i_{j+1}]$. Somit ist die Laufzeit aller Best-Aufrufe im worst-case proportional zur Anzahl der Elemente in $\bigcup_{j=j_0}^{n} [i_{j-1} : i_j] = [1 : n]$. Da aber nur die Elemente i_j dabei mehrfach vorkommen können und beim mehrmaligen Vorkommen jeweils j inkrementiert wird, ist die Laufzeit im worst-case linear und wir haben das folgende Lemma bewiesen.

Lemma 1.48 WMASS *kann in linearer Zeit gelöst werden, vorausgesetzt, die Funktion φ kann in konstanter Zeit berechnet werden.*

1.6.4 Linksschiefe Folgen und steigend linksschiefe Partitionen

Zur Lösung des Problems müssen wir nur noch zeigen, wie sich die Funktion φ effizient berechnen lässt. Hierzu benötigen wir den Begriff einer linksschiefen Folge sowie einer steigend linksschiefen Partition.

Definition 1.49 *Eine Folge* $(a_1, \ldots, a_n) \in \mathbb{R}^n$ *mit Gewichten* $(w_1, \ldots, w_n) \in \mathbb{N}^n$ *heißt* linksschief*, wenn für alle* $i \in [1 : n-1]$ *gilt:* $\mu(1, i) \geq \mu(i+1, n)$*.*

Eine Partition $a = A_1 \cdots A_k$ *von* $a = (a_1, \ldots, a_n)$ *heißt* steigend linksschief*, wenn jedes Segment* A_i *linksschief ist und* $\mu(A_i) < \mu(A_j)$ *für alle* $i < j$ *gilt.*

Anschaulich heißt eine Folge $a = (a_1, \ldots, a_n)$ also linksschief, wenn der Durchschnittswert jedes echten Präfix $(a_1, \ldots, a_i)$ größer gleich dem Durchschnittswert des korrespondierenden Suffixes $(a_{i+1}, \ldots, a_n)$ ist.

Lemma 1.50 *Sei* $a \in \mathbb{R}^n$ *bzw.* $b \in \mathbb{R}^m$ *mit Gewicht* $w \in \mathbb{N}^n$ *bzw.* $w' \in \mathbb{N}^m$ *jeweils eine linksschiefe Folge mit* $\mu(a) \geq \mu(b)$*. Dann ist auch die Folge* ab *mit Gewicht* ww' *linksschief.*

Beweis: Sei p ein beliebiges Präfix von ab. Es ist zu zeigen, dass $\mu(p) \geq \mu(q)$ ist, wobei q das zu p korrespondierende Suffix in ab ist, d.h. $ab = pq$.

Fall 1: Sei $p = a$. Dann gilt $\mu(a) \geq \mu(b)$ nach Voraussetzung, und die Behauptung ist erfüllt, da $q = b$ ist.

Fall 2: Sei jetzt p ein echtes Präfix von a, d.h. $a = pa'$. Da a eine linksschiefe Folge ist, gilt $\mu(p) \geq \mu(a')$. Mit dem Korollar 1.44 gilt dann:

$$\mu(p) \geq \mu(pa') \geq \mu(a').$$

Somit ist $\mu(p) \geq \mu(a) \geq \mu(b)$.

Mit $\mu(p) \geq \mu(a')$ und $\mu(p) \geq \mu(b)$, folgt

$$\begin{aligned} \mu(p) &= \frac{\ell(a')\mu(p) + \ell(b)\mu(p)}{\ell(a'b)} \\ &\geq \frac{\ell(a')\mu(a') + \ell(b)\mu(b)}{\ell(a'b)} \\ &= \frac{\sigma(a') + \sigma(b)}{\ell(a'b)} \end{aligned}$$

$$\begin{aligned} &= \frac{\sigma(a'b)}{\ell(a'b)} \\ &= \mu(a'b) \\ &= \mu(q). \end{aligned}$$

Fall 3: Sei $p = ab'$ mit $b = b'q$. Mit dem Korollar 1.44 folgt, da $b = b'q$ linksschief ist:

$$\mu(b') \geq \mu(\underbrace{b'q}_{=b}) \geq \mu(q).$$

Somit gilt $\mu(a) \geq \mu(b) \geq \mu(q)$. Also gilt mit $\mu(a) \geq \mu(q)$ und $\mu(b') \geq \mu(q)$:

$$\begin{aligned} \mu(p) &= \mu(ab') \\ &= \frac{\sigma(ab')}{\ell(ab')} \\ &= \frac{\sigma(a) + \sigma(b')}{\ell(ab')} \\ &= \frac{\ell(a)\mu(a) + \ell(b')\mu(b')}{\ell(ab')} \\ &\geq \frac{\ell(a)\mu(q) + \ell(b')\mu(q)}{\ell(ab')} \\ &= \mu(q). \end{aligned}$$

Damit ist das Lemma bewiesen. ■

Lemma 1.51 *Jede Folge $a \in \mathbb{R}^n$ mit Gewichten $w \in \mathbb{N}^n$ besitzt eine eindeutig steigend linksschiefe Partition.*

Beweis: Wir führen den Beweis durch Induktion über n.

Induktionsanfang ($n = 1$): Klar, nach Definition.

Induktionsschritt ($n \to n + 1$): Wir betrachten eine Folge $(a_1, \ldots, a_n, a_{n+1})$. Sei weiter $A_1 \cdots A_k$ die steigend linksschiefe Partition von $(a_1, \ldots, a_n)$.

Gilt $\mu(A_k) < a_{n+1}/w_{n+1} = \mu(a_{n+1})$, dann ist $A_1 \cdots A_k \cdot (a_{n+1})$ eine neue steigend linksschiefe Partition.

Andernfalls bestimmen wir ein maximales i mit $\mu(A_{i-1}) < \mu(A_i \cdots A_k \cdot (a_{n+1}))$. Dann behaupten wir, dass die neue Partition $A_1 \cdots A_{i-1} \cdot (A_i \cdots A_k \cdot (a_{n+1}))$ eine steigend linksschiefe ist.

Nach Definition ist (a_{n+1}) linksschief. Nach dem vorherigen Lemma 1.50 und der Wahl von i ist dann auch $A_k \cdot (a_{n+1})$ linksschief. Weiter gilt allgemein für $j \geq i$, dass $A_j \cdots A_k \cdot (a_{n+1})$ linksschief ist. Somit ist auch $A_i \cdots A_k \cdot (a_{n+1})$ linksschief.

Nach Konstruktion ist also die jeweils konstruierte Partition eine steigend linksschiefe Partition.

Es bleibt noch die Eindeutigkeit zu zeigen. Nehmen wir an, es gäbe zwei verschiedene steigend linksschiefe Partitionen. Betrachten wir, wie in der folgenden Abbildung 1.45 skizziert, die jeweils linkesten Teilfolgen in ihren Partition, in denen sich die beiden Partitionen unterscheiden.

Abbildung 1.45: Skizze: Beweis der Eindeutigkeit der linksschiefen Partition

Betrachten wir zuerst den Fall P_1 gegen P_2 Sei zuerst $z = \varepsilon$. Dann gilt $\mu(x) < \mu(y)$, da P_2 eine steigend linksschiefe Partition ist. Da xy in P_1 linksschief ist, gilt weiterhin $\mu(x) \geq \mu(y)$ und wir erhalten den gewünschten Widerspruch.

Sei nun $z \neq \varepsilon$. Da yz nach der Partition P_2 linksschief ist, gilt $\mu(y) \geq \mu(z)$. Mit Korollar 1.44 folgt, dass $\mu(y) \geq \mu(yz) \geq \mu(z)$.

Nach Wahl der steigend linksschiefen Partition P_2 gilt $\mu(x) < \mu(yz)$. Wie wir eben gezeigt haben, gilt auch $\mu(yz) \leq \mu(y)$. Damit ist $\mu(x) < \mu(y)$ und somit ist xy nicht linksschief. Dies ist ein Widerspruch zur Annahme, dass P_1 eine steigend linksschiefe Partition ist.

Es bleibt noch der Fall P_1 gegen P_3 zu betrachten. Nach P_1 gilt $\mu(x) \geq \mu(y)$ und nach P_3 gilt $\mu(x) < \mu(y')$. Also gilt $\mu(y) < \mu(y')$. Sei y'' so gewählt, dass $y = y'y''$ und sei weiter $y = y' \cdot Y_2 \cdots Y_k$ eine steigend linksschiefe Partition von y, die sich aus P_3 ergibt. Dann gilt $\mu(y') < \mu(Y_2) < \cdots < \mu(Y_k)$. Dann muss aber $\mu(y') < \mu(y'')$ sein. Somit gilt $\mu(y) < \mu(y') < \mu(y'')$, was ein offensichtlicher Widerspruch ist. ■

Wir merken noch an, dass sich aus diesem Beweis unmittelbar ein Konstruktionsalgorithmus für die steigende linksschiefe Partition einer gegebenen gewichteten Folge ergibt. Die Laufzeit ist auch hier wieder linear in der Länge der Folge.

Korollar 1.52 *Für eine Folge $a \in \mathbb{R}^n$ mit Gewichten $w \in \mathbb{N}^n$ kann ihre steigend linksschiefe Partition in Zeit $O(n)$ berechnet werden.*

Beweis: Wir müssen nur die Laufzeit beweisen. Für jedes neues Folgenglied a_i wird ein neues Segment generiert. Die Anzahl der benötigten Verschmelzungen kann insgesamt nicht größer als die Anzahl der insgesamt generierten Segmente sein. Da diese $O(n)$ ist, kann es auch nur $O(n)$ Verschmelzungen geben und somit ist die Gesamtlaufzeit linear. ■

Das folgende Lemma zeigt, wie uns die steigende linksschiefe Partition für die Berechnung der Funktion φ in den benötigten Fällen hilft.

Lemma 1.53 *Sei $i \leq r \leq j \in \mathbb{N}$ und sei $(a_i, \ldots, a_j) \in \mathbb{R}^{j-i+1}$ eine reelle Folge mit Gewichten $(w_i, \ldots, w_j) \in \mathbb{N}^{j-i+1}$. Weiter bezeichne $p(m)$ die Endposition des Segments einer steigend linksschiefen Partition von $(a_i, \ldots, a_{r-1})$, die an Position m beginnt. Dann gilt $\varphi(i, r-1) = p(i)$.*

Beweis: Sei $A_1 \cdots A_k$ die steigend linksschiefe Partition von $(a_i, \ldots, a_{r-1})$. Für einen Widerspruchsbeweis nehmen wir an, dass $\varphi(i, r-1) \neq p(i)$.

Fall 1 ($\boldsymbol{\varphi(i, r-1) > p(i)}$): Da für jedes $m \in [1:k]$ das Segment A_m linksschief ist, gilt für jedes Präfix A'_m von A_m, dass $\mu(A'_m) \geq \mu(A_m)$. Da $\mu(A_1) < \mu(A_m)$ für alle $m \in [2:k]$ (steigend linksschiefe Partition), gilt mit Lemma 1.43 für ein $m \in [2:k]$ also

$$\mu(i, \varphi(i, r-1) = \mu(A_1 \cdot A_2 \cdots A_{m-1} \cdot A'_m) > \mu(A_1) = \mu(i, p(i))$$

Dies ist ein Widerspruch zur Definition von φ.

Fall 2 ($\boldsymbol{\varphi(i, r-1) < p(i)}$): Dann ist $(a_i, \ldots, a_{\varphi(i,r-1)})$ ein Präfix von A_1. Da A_1 linksschief ist, gilt $\mu(i, \varphi(i, r-1) \geq \mu(\varphi(i, r-1)+1, p(i))$. Aufgrund von Korollar 1.44 gilt dann allerdings auch $\mu(i, \varphi(i, r-1) \geq \mu(i, p(i))$, was ein Widerspruch zur Definition von φ ist. ■

In Abbildung 1.46 ist der so modifizierte Algorithmus zur Lösung von WMASS angegeben. Beachte hierbei, dass p nur partiell definiert ist. Für $i \in [i_{j-1} : r_j - 1]$

```
WMASS (real a[], real w[], int n, int λ)
begin
    int opt := 0;
    int L := 0;
    int R := 0;
    int p[1 : n] ;                                        /* not initialized */
    int i_{j0-1} := 1;
    for (j := j0; j ≤ n; j++) do
        update increasing leftskew partition p of (a_{i_{j-1}}, ..., a_{r_j - 1});
        i_j := Best(i_{j-1}, r_j, j);
        if (μ(i_j, j) > opt) then
            opt := μ(i_j, j);
            L := i_j;
            R := j;
end

Best(ℓ, r, j)
begin
    int i := ℓ;
    while ((i < r) && (μ(i, p(i)) ≤ μ(i, j))) do
        i := p(i) + 1;
    return i;
end
```

Abbildung 1.46: Algorithmus: WMASS

gibt $p(i)$ die Endposition des Segments, das an Position i beginnt, einer steigend linksschiefen Partition von $(a_{i_{j-1}}, \ldots, a_{r_j-1})$. Für alle anderen Werte ist $p(i)$ entweder undefiniert oder enthält einen Wert, der ohne Bedeutung ist.

Die Prozedur Update wird konsekutiv für die Teilfolgen $(a_{i_{j-1}}, \ldots, a_{r_j-1})$ aufgerufen, wobei wir immer schon für einen Präfix der Folge (eventuelle auch einem leeren, wie zu Beginn) die eindeutige steigende linksschiefe Partition bestimmt haben. Wir müssen diese also nur noch ergänzen, wie im Lemma 1.51 beschrieben. Somit bestimmen wir mittels Update Stück für Stück die eindeutige steigende linksschiefe Partion von a, die sich in linearer Zeit berechnen lässt. Somit sind die zusätzlichen Berechnungskosten zum generischen Algorithmus $O(n)$ und wir erhalten folgenden Satz.

Theorem 1.54 *Für eine Folge $a \in \mathbb{R}^n$ mit Gewichten $w \in \mathbb{N}^n$ kann* WMASS *in Zeit $O(n)$ gelöst werden.*

12.11.20

Suffix Trees Revisited 2

2.1 Definition von Suffix Tries und Suffix Trees

In diesem Kapitel betrachten wir einige Algorithmen zur Konstruktion von Suffix Tries bzw. Suffix Trees. Letztere werden zur effizienten Suche von Wörtern und Teilstrings (z.B. Tandem-Repeats) im nächsten Kapitel benötigt.

2.1.1 Σ-Bäume und (kompakte) Σ^+-Bäume

Zunächst definieren wir die so genannten Σ-Bäume.

Definition 2.1 *Sei Σ ein Alphabet. Ein* Σ-Baum *ist ein gewurzelter Baum mit Kantenmarkierungen aus Σ, so dass kein Knoten zwei ausgehende Kanten mit derselben Markierung besitzt. Ein Σ-Baum wird oft auch als* Trie *bezeichnet.*

In Abbildung 2.1 ist ein Beispiel eines Σ-Baumes mit $\Sigma = \{a, b\}$ angegeben.

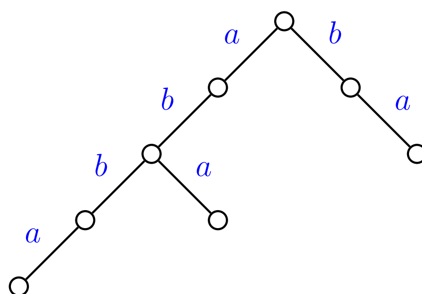


Abbildung 2.1: Beispiel: Ein Σ-Baum T_1 mit $\Sigma = \{a, b\}$

Erlaubt man als Kantenmarkierung Zeichenreihen über Σ, so erhält man die so genannten Σ^+-Bäume.

Definition 2.2 *Sei Σ ein Alphabet. Ein* Σ^+-Baum *ist ein gewurzelter Baum mit Kantenmarkierungen aus Σ^+, so dass kein Knoten zwei ausgehende Kanten besitzt, deren Markierungen mit demselben Zeichen aus Σ beginnen.*

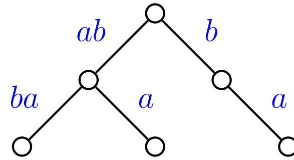


Abbildung 2.2: Beispiel: Ein Σ^+-Baum T_2 mit $\Sigma = \{a, b\}$

In Abbildung 2.2 ist ein Beispiel eines Σ^+-Baumes mit $\Sigma = \{a, b\}$ angegeben.

Werden in Σ^+-Bäumen Knoten mit nur einem Kind verboten, so erhalten wir kompakte Σ^+-Bäume.

Definition 2.3 *Ein Σ^+-Baum heißt* kompakt, *wenn es keinen Knoten außer der Wurzel mit nur einem Kind gibt. Ein solcher kompakter Σ^+-Baum wird auch als* kompaktifizierter Trie *bezeichnet.*

In Abbildung 2.3 ist ein Beispiel eines kompakten Σ^+-Baumes mit $\Sigma = \{a, b\}$ angegeben.

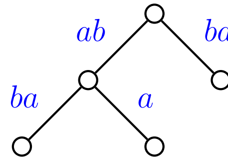


Abbildung 2.3: Beispiel: Ein kompakter Σ^+-Baum T_3 mit $\Sigma = \{a, b\}$

2.1.2 Grundlegende Notationen und elementare Eigenschaften

In diesem Abschnitt wollen wir noch einige grundlegende Notationen und Eigenschaften von Σ^+-Bäumen einführen.

Sei im Folgenden T ein Σ^+-Baum.

- Für einen Knoten $v \in V(T)$ bezeichnen wir mit $\text{path}(v)$ die Konkatenation der Kantenmarkierungen auf dem Pfad von der Wurzel zu v, siehe auch Abbildung 2.4, und $|\text{path}(v)|$ wird als *Worttiefe* von v bezeichnet.
- Wir sagen, dass eine Zeichenreihe $x \in \Sigma^*$ von T *dargestellt* wird, wenn es einen Knoten $v \in V(T)$ und ein $y \in \Sigma^*$ gibt, so dass $\text{path}(v) = xy$.

  In Abbildung 2.4 stellt der Σ^+-Baum unter anderen das Wort *abb* dar.

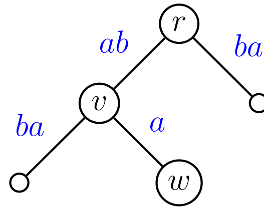


Abbildung 2.4: Beispiel: $\text{path}(v) = ab$ und $\text{path}(w) = aba$ in T_3

- Gilt $\text{path}(v) = x$ für einen Knoten $v \in V(T)$ und eine Zeichenreihe $x \in \Sigma^*$, dann schreiben wir $\overline{x}$ für v.

  In Abbildung 2.4 gilt im angegebenen Σ^+-Baum beispielsweise $\overline{ab} = v$ und $\overline{\varepsilon} = r$ bezeichnet die Wurzel dieses Baumes.

- Mit $\text{words}(T)$ bezeichnen wir die Menge der Wörter, die im Σ^+-Baum T dargestellt werden. Formal lässt sich das wie folgt definieren:

  $$\text{words}(T) = \{x \,:\, \exists v \in V(T), y \in \Sigma^* : \text{path}(v) = xy\}\,.$$

  In Abbildung 2.4 gilt für den Σ^+-Baum T beispielsweise

  $$\text{words}(T) = \{\varepsilon, a, ab, aba, abb, abba, b, ba\},$$

  wobei ε wie üblich das leere Wort bezeichnet.

  Für die Bäume aus den Abbildungen 2.1, 2.2 und 2.3 gilt

  $$\text{words}(T_1) = \text{words}(T_2) = \text{words}(T_3).$$

2.1.3 Suffix Tries und Suffix Trees

Bevor wir zu Suffix Tries und Suffix Trees kommen, wiederholen wir erst noch die Definitionen und zugehörigen Notationen zu Präfixen, Suffixen und Teilwörtern.

Notation 2.4 *Sei Σ ein Alphabet und $w \in \Sigma^*$. Sei weiter v ein* Teilwort *von w (d.h. es gibt $x, y \in \Sigma^*$ mit $w = xvy$), dann schreiben wir auch $v \sqsubseteq w$.*

Beobachtung 2.5 *Seien $v, w \in \Sigma^*$ und $¢, \$ \notin \Sigma$, dann ist v genau dann ein Präfix (bzw. Suffix) von w, wenn $¢v \sqsubseteq ¢w$ (bzw. $v\$ \sqsubseteq w\$$) gilt.*

Damit kommen wir zur Definition eines Suffix Tries.

Definition 2.6 *Sei Σ ein Alphabet. Ein* Suffix Trie *für ein Wort $t \in \Sigma^*$ ist ein Σ-Baum T mit der Eigenschaft $\text{words}(T) = \{w \in \Sigma^* \,:\, w \sqsubseteq t\}$.*

In Abbildung 2.5 ist der Suffix Trie für das Wort *abbab* angegeben.

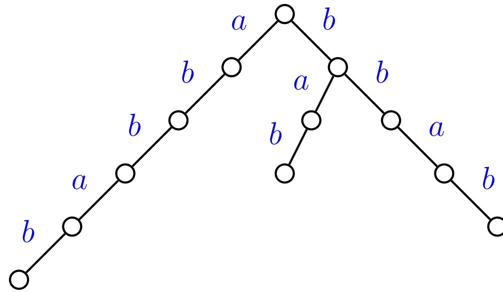


Abbildung 2.5: Beispiel für einen Suffix Trie mit $t = abbab$

Kommen wir nun zur Definition eines Suffix-Baumes.

Definition 2.7 *Sei* Σ *ein Alphabet und* $t \in \Sigma^*$*. Ein* Suffix-Baum *(engl.* Suffix Tree*) für* t *ist ein kompakter* Σ^+*-Baum* $T = T(t)$ *mit* $words(T) = \{w \in \Sigma^* : w \sqsubseteq t\}$*.*

In Abbildung 2.6 ist links ein Suffix-Baum für das Wort *abbab* angegeben.

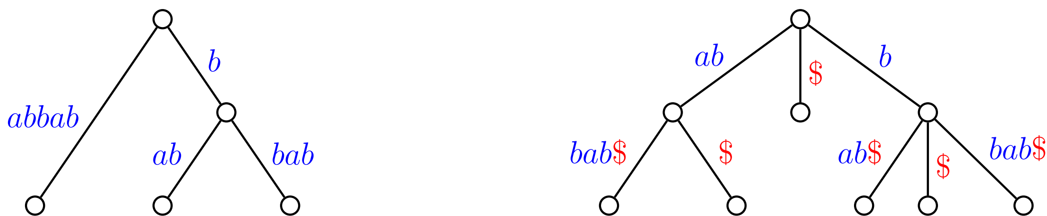


Abbildung 2.6: Beispiel: Suffix-Bäume für $t = abbab$ und $t = abbab\$$

Warum diese Bäume Suffix Tries bzw. Suffix Trees statt Infix Tries bzw. Infix Trees heißen, wird gleich in den folgenden Bemerkungen klar.

Bemerkungen: Halten wir zunächst ein paar nützliche Eigenschaften von Suffix Tries bzw. -Trees fest.

- Oft betrachtet man statt des Suffix Tries bzw. Suffix Trees für $t \in \Sigma^*$ den Suffix Trie bzw. Suffix Tree für $t\$ \in (\Sigma \cup \{\$\})^*$, wobei $\$ \notin \Sigma$ gilt. Der Vorteil dieser Betrachtungsweise ist, dass dann jedes Suffix von t zu einem Blatt des Suffix Tries bzw. Suffix Trees für t korrespondiert. Damit wird dann auch die Bezeichnung Suffix Trie bzw. Suffix Tree klar.

- Als Kantenmarkierungen werden keine Zeichenreihen verwendet, sondern so genannte *Referenzen* (Start- und Endposition) auf das Originalwort. Für ein Beispiel siehe auch Abbildung 2.7. Statt ab wird dort die Referenz $(1, 2)$ für t_1t_2 im Wort $t = abbab\$$ angegeben. Man beachte, dass ab mehrfach in t vorkommt, nämlich ab Position 1 und ab Position 4. Die Wahl der Referenz wird dabei nicht willkürlich sein. Wir betrachten dazu die Kante mit Kantenmarkierung w, deren Referenz bestimmt werden soll. Im Unterbaum des Suffix-Baumes, der am unteren Ende der betrachteten Kante gewurzelt ist, wird das längste im Unterbaum dargestelle Suffix s betrachtet. Dann muss auch $w \cdot s$ ein Suffix von t sein und als Referenz für die Kantenmarkierung w wird $(|t| - |w \cdot s| + 1, |t| - |s|)$ gewählt.

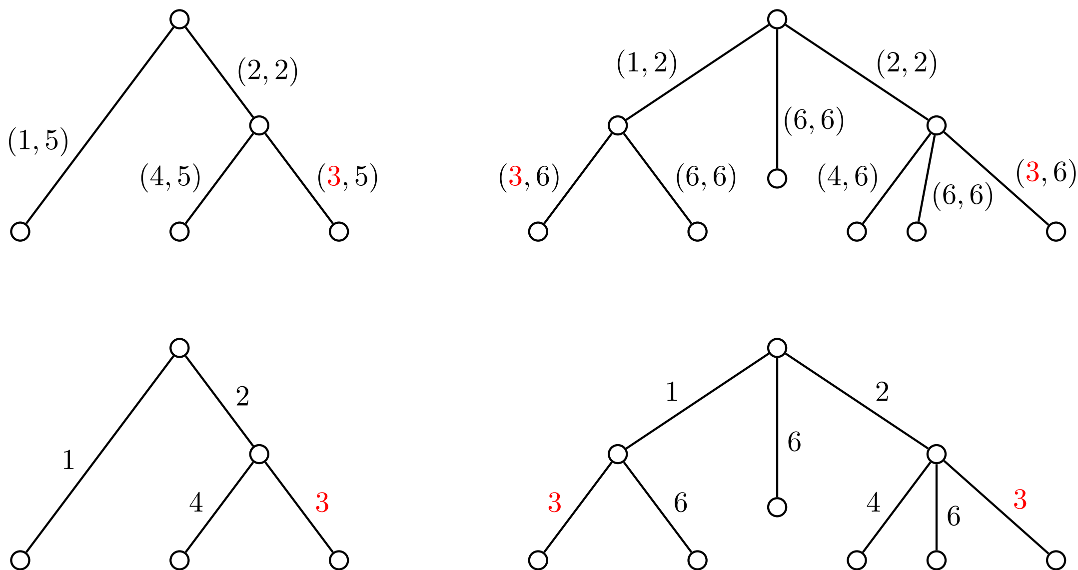


Abbildung 2.7: Beispiel: Suffix-Bäume für $t = abbab$ bzw. $t = abbab\$$ (links bzw. rechts) mit normalen (oben) bzw. reduzierten (unten) Referenzen als Kantenmarkierungen

- Der Platzbedarf für einen Suffix-Baum beträgt mit dieser Referenzdarstellung $O(|t|)$. Würde man hingegen die Zeichenreihen an die Kanten schreiben, so könnte der Platzbedarf auch auf $\Theta(|t|^2)$ ansteigen. Der Beweis ist dem Leser als Übungsaufgabe überlassen.

- In der Literatur wird der Suffix Trie manchmal auch als *atomic suffix tree* und der Suffix Tree selbst als *compact suffix tree* bezeichnet. Dies führt oft zu Verwirrung, insbesondere, da heutzutage oft unter einem kompakten Suffix-Baum (compact suffix tree) etwas ganz anderes verstanden wird.

2.2 Repräsentationen von Bäumen

Zuerst überlegen wir uns einige Möglichkeiten, wie man Bäume im Allgemeinen und Suffix-Bäume im Speziellen überhaupt speichern kann. In einem Baum muss man im Wesentlichen die Kinder eines Knoten speichern. Dafür gibt es verschiedene Möglichkeiten, die jeweils ihre Vor- und Nachteile haben.

Allgemein erinnern wir uns zuerst daran, dass ein Baum mit m Blättern, der keinen Knoten mit nur einem Kind besitzt, maximal $m-1$ innere Knoten haben kann. Somit besitzt ein Suffix-Baum für t weniger als $|t|$ innere Knoten und somit insgesamt weniger als $2|t|$ Knoten. Der Leser möge sich überlegen, was passiert, wenn in einem Suffix-Baum die Wurzel nur ein Kind besitzt.

Für eine detailliertere Analyse in der Bioinformatik nehmen wir an, dass sowohl Integers als auch Verweise für Ihre Speicherung 4 Bytes benötigen (32-Bit-Architektur). Bei Annahme einer 64-Bit-Architektur wären es hingegen 8 Bytes und in den folgenden detaillierte Analysen verdoppelt sich der Speicherbedarf in Bytes in etwa. Für die Kantenmarkierungen werden wir in der Regel nicht die volle Referenz (i, j) speichern, sondern oft nur die Startposition i. Da in einem Suffixbaum alle Suffixe gespeichert sind, muss von dem Knoten, den wir über die Kante (i, j) erreichen, eine Kante mit der Startposition $j + 1$ beginnen. Wenn wir uns diese merken (meist als erste ausgehende Kante in unseren Darstellungen), können wir daraus die Endposition j rekonstruieren. Hierfür ist es wichtig, dass wir für ein Teilwort w von t die Referenz geeignet wählen (d.h. wie im vorherigen Abschnitt beschrieben).

Im Folgenden ist bei der Zeitanalyse der verschiedenen Darstellungen von Bäumen mit Zeit die Zugriffszeit auf ein Kind gemeint, zu dem die Kantenmarkierung mit einem bestimmten Zeichen beginnt.

2.2.1 Darstellung der Kinder mit Feldern

Die Kinder eines Knotens lassen sich sehr einfach mit Hilfe eines Feldes der Größe $|\Sigma|$ darstellen.

- Platz: $O(|t| \cdot |\Sigma|)$.

  Dies folgt daraus, dass für jeden Knoten ein Feld mit Platzbedarf $O(|\Sigma|)$ benötigt wird.

- Zeit: $O(1)$.

  Die üblichen Realisierungen von Feldern erlauben einen Zugriff in konstanter Zeit auf die einzelnen Feldelemente.

Der Zugriff ist also sehr schnell, wo hingegen der Platzbedarf, insbesondere bei großen Alphabeten, doch sehr groß werden kann.

Für eine DNA-Sequenz sind also pro Knoten maximal 20 Bytes nötig (16 Bytes für die Verweise und 4 Bytes für die Referenz auf die Kantenmarkierung). Da es etwa doppelt so viele Knoten wie Blätter (also Nukleotide in der gegebenen DNA-Sequenz) gibt, sind als pro Nukleotid maximal 40 Bytes nötig.

Für ein Protein sind also pro Knoten maximal 84 Bytes nötig (80 Bytes für die Verweise und 4 Bytes für die Referenz auf die Kantenmarkierung). Da es etwa doppelt so viele Knoten wie Blätter (also Aminosäuren im gegebenen Protein) gibt, sind als pro Aminosäure maximal 168 Bytes nötig.

Für die zweite Referenz müssen wir uns noch explizit das Kind merken, deren Referenz im Wesentlichen das Ende der gesuchten Referenz angibt. Dazu sind im Falle von DNA bzw. Proteinen noch jeweils 2 Bits bzw. 5 Bits je inneren Knoten nötig (wenn man die Kinder in konstanter Zeit finden will).

Insgesamt sind daher pro Nukleotid bzw. pro Aminosäure etwa 42 Bytes bzw. etwa 173 Bytes nötig.

2.2.2 Darstellung der Kinder mit Listen

Eine andere Möglichkeit ist, die Kinder eines Knotens in einer linearen Liste zu verwalten, wobei das erste Kind einer Liste immer dasjenige ist, das für den zweiten Referenzwert der inzidenten Kante zu seinem Elter nötig ist.

- Platz: $O(|t|)$.

  Für jeden Knoten ist der Platzbedarf proportional zur Anzahl seiner Kinder. Damit ist Platzbedarf insgesamt proportional zur Anzahl der Knoten des Suffix-Baumes, da jeder Knoten (mit Ausnahme der Wurzel) das Kind eines Knotens ist. Im Suffix-Baum gilt, dass jeder Knoten entweder kein oder mindestens zwei Kinder hat. Für solche Bäume ist bekannt, dass die Anzahl der inneren Knoten kleiner ist als die Anzahl der Blätter. Da ein Suffix-Baum für einen Text der Länge m maximal m Blätter besitzt, folgt daraus die Behauptung für den Platzbedarf.

- Zeit: $O(|\Sigma|)$.

  Leider ist hier die Zugriffszeit auf ein Kind sehr groß, da im schlimmsten Fall (aber größenordnungsmäßig auch im Mittel) die gesamte Kinderliste eines Knotens durchlaufen werden muss und diese bis zu $|\Sigma|$ Elemente umfassen kann.

In vielen Anwendungen wird jedoch kein direkter Zugriff auf ein bestimmtes Kind benötigt, sondern eine effiziente Traversierung aller Kinder. Dies ist mit der Geschwisterliste aber auch wieder in konstanter Zeit pro Kind möglich, wenn keine besondere Ordnung auf den Kindern berücksichtigt werden soll.

Pro Knoten des Baumes sind also maximal 12 Bytes nötig (8 Bytes für die Verweise und 4 Bytes für die Kantenmarkierung). Für die gespeicherte Sequenz sind also 24 Bytes pro Element nötig. Bei geschickterer Darstellung der Blätter (hier ist ja kein Verweis auf sein ältestes Kind nötig) kommen wir mit 20 Bytes aus.

Wenn man eine Ordnung in den Listen aufrecht erhalten will, so muss man auf die reduzierten Referenzen verzichten und pro Nukleotid nochmals 4–8 Bytes opfern (je nachdem, wie geschickt man die Referenzen an den zu Blättern inzidenten Kanten implementiert).

2.2.3 Darstellung der Kinder mit balancierten Bäumen

Die Kinder lassen sich auch mit Hilfe von balancierten Suchbäumen (AVL-, Rot-Schwarz-, B-Bäume, etc.) verwalten:

- Platz: $O(|t|)$

  Da der Platzbedarf für einen Knoten ebenso wie bei linearen Listen proportional zur Anzahl der Kinder ist, folgt die Behauptung für den Platzbedarf unmittelbar.

- Zeit: $O(\log(|\Sigma|))$.

  Da die Tiefe von balancierten Suchbäumen logarithmisch in der Anzahl der abzuspeichernden Schlüssel ist, folgt die Behauptung unmittelbar.

  Auch hier ist eine Traversierung der Kinder eines Knotens in konstanter Zeit pro Kind möglich, indem man einfach den zugehörigen balancierten Baum traversiert.

Auch hier sind für eine Sequenz also pro Element maximal etwa 24-26 Bytes nötig. Dies folgt daraus, das balancierte Bäume etwa genauso viele Links besitzen wie Listen (wenn man annimmt, dass die Daten auch in den inneren Knoten gespeichert werden und man auf die unnötigen Referenzen in den Blättern verzichtet, sonst erhöht sich der Platzbedarf jeweils um etwa einen Faktor 2). Es werden jedoch noch zusätzlich 1–2 Bytes für die Verwaltung der Balancierung benötigt.

2.2.4 Darstellung des Baumes mit einer Hash-Tabelle

Eine weitere Möglichkeit ist die Verwaltung der Kinder aller Knoten in einem einzigen großen Feld der Größe $O(|t|)$. Um nun für ein Knoten auf ein spezielles Kind zuzugreifen wird dann eine *Hashfunktion* verwendet:

$$h : V \times \Sigma \to \mathbb{N} : (v, a) \mapsto h(v, a).$$

Hierbei interpretieren wir die Knotenmenge V in geeigneter Weise als natürliche Zahlen, d.h. $V \subset \mathbb{N}$. Zu jedem Knoten und dem Symbol, die das Kind identifizieren, wird ein Index des globalen Feldes bestimmt, an der die gewünschte Information enthalten ist (also hier die entsprechende Kantenmarkierung).

Leider bilden Hashfunktionen ein relativ großes Universum von potentiellen Referenzen (hier Paare von Knoten und Symbolen aus Σ, also mit einer Mächtigkeit von $\Theta(|t| \cdot |\Sigma|)$) auf ein kleines Intervall ab (hier Indizes aus $[1 : \ell]$ mit $\ell = \Theta(|t|)$). Daher sind so genannte *Kollisionen* prinzipiell nicht auszuschließen. Ein Beispiel ist das so genannte *Geburtstagsparadoxon*. Ordnet man jeder Person in einem Raum eine Zahl aus dem Intervall $[1 : 366]$ zu (nämlich ihren Geburtstag), dann ist bereits ab 23 Personen die Wahrscheinlichkeit größer als 50%, dass zwei Personen denselben Wert erhalten. Also muss man beim Hashing mit diesen Kollisionen leben und diese geeignet auflösen. Für die Details der Kollisionsauflösung verweisen wir auf andere Vorlesungen.

Um solche Kollisionen überhaupt festzustellen, enthält jeder Feldeintrag i neben den normalen Informationen noch die Informationen, wessen Kind er ist und über welches Symbol er von seinem Elter erreichbar ist. Somit lassen sich Kollisionen leicht feststellen und die üblichen Operationen zur Kollisionsauflösung anwenden.

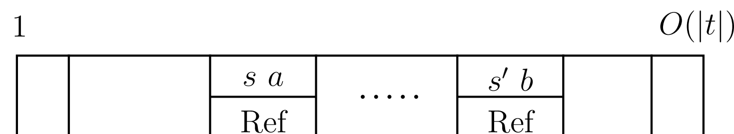


Abbildung 2.8: Skizze: Realisierung mittels eines Feldes und Hashing

- Platz: $O(|t|)$

  Das folgt unmittelbar aus der obigen Diskussion.

- Zeit: $O(1)$ (bei einer perfekten Hashfunktion)

  Im Wesentlichen erfolgt der Zugriff in konstanter Zeit, wenn man voraussetzt, dass sich die Hashfunktion einfach (d.h. in konstanter Zeit) berechnen lässt und dass sich Kollisionen effizient auflösen lassen.

Hier sind für jeden Eintrag 4 Bytes für die Kantenmarkierung nötig. Für die Verwaltung (ein Verweis auf den Elter und das Zeichen, das das Kind identifiziert) sind 5 Bytes nötig. Um die Endposition zu merken, muss man sich noch das entsprechende Kind merken, was ein weiteres Byte kostet (für das Zeichen, über das das Kind erreicht wird). Da es maximal $2|t|$ Kanten gibt, sind somit maximal 20 Bytes pro Element nötig. Da beim Hashing für eine effiziente Kollisionsauflösung das Feld nur bis zu etwa 80% gefüllt sein darf, muss das Feld entsprechend größer gewählt werden, so dass der Speicherbedarf insgesamt maximal 25 Bytes pro Element der Sequenz beträgt.

2.2.5 Speicherplatzeffiziente Feld-Darstellung

Man kann die Methode der linearen Listen auch selbst mit Hilfe eines Feldes implementieren. Dabei werden die Geschwisterlisten als konsekutive Feldelemente abgespeichert. Begonnen wird dabei mit der Geschwisterliste des ältesten Kindes der Wurzel, also mit den Kindern der Wurzel, die dann ab Position 1 des Feldes stehen. Die Wurzel selbst wird dabei nicht explizit dargestellt.

Auch hier unterscheidet man wieder zwischen internen Knoten und Blättern. Ein interner Knoten belegt zwei Feldelemente, ein Blatt hingegen nur ein Feldelement. Ein innerer Knoten hat einen Verweis auf das Feldelement, in dem die Geschwisterliste des ältesten Kindes konsekutiv abgespeichert wird. Im zweiten Feldelement steht die Startposition der Kantenmarkierung innerhalb von t für die Kante, die in den Knoten hineinreicht. Für ein Blatt steht nur die Startposition der Kantenmarkierung innerhalb von t für die Kante, die in das Blatt hineinreicht.

Ein Beispiel ist in Abbildung 2.9 für $t = abbab\$$ angegeben. Zur Unterscheidung der beiden Knotentypen wird ein Bit spendiert. Dies ist in der Abbildung 2.9 als hochgestelltes B für die Blätter zu erkennen (die anderen sind innere Knoten). Des Weiteren müssen mit Hilfe eines weiteren Bits auch noch die Enden der Geschwisterlisten markiert werden. Dies ist in Abbildung 2.9 mit einem tiefergestellten $*$ markiert. Verweise sind dort als rote Zahlen geschrieben.

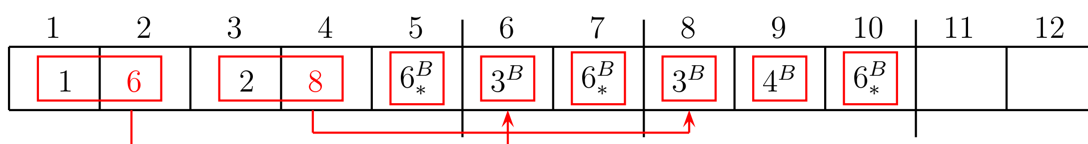


Abbildung 2.9: Beispiel: Feld-Darstellung des Suffix-Baumes aus Abbildung 2.6 für $t = abbab\$$

Es bleibt nur die Frage, wie man das Ende einer Kantenmarkierung findet. Das ist jedoch sehr einfach: Hat man den Beginn der Kantenmarkierung, so folgt man dem

Verweis dieses Knotens. Im folgenden Knoten steht dann der Beginn der nächsten Kantenmarkierung; dieser muss nur noch um es erniedrigt werden. Handelt es sich um ein Blatt, so gibt es keinen Folgeverweis, aber die Endposition ist dann das Ende, also $|t|$. Hierfür ist aber wichtig, dass die Knoten der Geschwisterliste in der richtigen Reihenfolge abgespeichert werden, nämlich das Kind, dessen zugehörige Kantenmarkierung am frühestens in t auftritt, als erstes.

- Platz: $O(|t|)$.

  Wie bei Listen

- Zeit: $O(|\Sigma|)$.

  Leider ist hier die Zugriffszeit auf ein Kind sehr groß, da im schlimmsten Fall (aber größenordnungsmäßig auch im Mittel) die gesamte Kinderliste eines Knotens durchlaufen werden muss und diese bis zu $|\Sigma|$ Elemente umfassen kann.

  Auch hier ist eine Traversierung der Kinder eines Knotens in konstanter Zeit pro Kind möglich, indem man einfach im Feld weiterläuft bis man auf eine Markierung stößt, die das Ende der Geschwisterliste markiert.

Pro inneren Knoten wird nun eine Zahl, ein Verweis und zwei Bits gespeichert, pro Blatt eine Zahl und zwei Bits. Spendiert man von den 4 Bytes jeweils zwei Bits um die Bits dort zu Markierung von Blättern und dem Ende von Geschwisterlisten zu speichern, kommt man mit maximal 12 Bytes pro Sequenzelement aus.

2.3 WOTD-Algorithmus

Nun wollen wir einen einfachen Algorithmus zur Konstruktion von Suffix-Bäumen angeben. Der so genannte *WOTD-Algorithmus* (für write-only-top-down) fügt die Kanten in den Suffix-Baum von der Wurzel beginnend (daher top-down) ein.

2.3.1 Die Konstruktion

Zuerst werden alle möglichen Suffixe des Wortes $t\$$ (mit $t \in \Sigma^*$ und $\$ \notin \Sigma$) bestimmt und hierfür der rekursive Algorithmus mit der Wurzel und dieser Menge von Suffixen aufgerufen. Die Menge von Suffixen wird dabei natürlich durch deren Startpositionen repräsentiert. Der rekursive Algorithmus konstruiert nun einen Baum mit der angegeben Wurzel, der alle Wörter in der übergebenen Menge darstellen soll.

```
WOTD (char t[])
begin
    set of words S(t) := {y : ∃x ∈ Σ* : xy = t$};       /* all suffixes of t$ */
    node r;                                              /* root of the suffix-tree */
    WOTD_REC(S(t), r);
end

WOTD_REC(set of words S, node v)
begin
    sort S according to the first character using bucket-sort;
    let S_c := {x ∈ S : x = c = $ ∨ ∃z ∈ Σ*$ : x = cz} for all c ∈ Σ ∪ {$};
    for (c ∈ Σ ∪ {$}) do
        if (|S_c| = 1) then
            append a new leaf to v with edge label w ∈ S_c;
        else if (|S_c| > 1) then
            // determine a longest common prefix in S_c
            let p be a longest word in {p' ∈ Σ* : ∀x ∈ S_c: ∃z ∈ Σ*$): x = p'z};
            append a new node w to v with edge label p;
            S'_c := p^{-1} · S_c := {z ∈ Σ*$ : ∃x ∈ S_c: x = pz};
            WOTD_REC(S'_c,w);
end
```

Abbildung 2.10: Algorithmus: WOTD (Write-Only-Top-Down)

Im rekursiven Algorithmus wird die übergebenen Menge von Wörtern nach dem ersten Buchstaben sortiert. Hierzu wird vorzugsweise ein Bucket-Sort verwendet. Rekursiv wird dann daraus der Suffix-Baum aufgebaut. Für den eigentlichen Aufbau traversiert man den zu konstruierenden Suffix-Baum top-down. Für jeden Knoten gibt es eine Menge S von Zeichenreihen (Suffixe von $t\$$), die noch zu verarbeiten sind. Diese wird dann nach dem ersten Zeichen (mittels eines Bucket-Sorts) sortiert und liefert eine Partition der Zeichenreihe in Mengen S_c mit demselben ersten Zeichen für $c \in \Sigma \cup \{\$\}$. Für jede nichtleere Menge S_c wird ein neues Kind generiert, wobei die inzidente Kante als Kantenmarkierung das längste gemeinsame Präfix p aus S_c erhält. Dann werden von jeder Zeichenreihe aus S_c dieses Präfix p entfernt und in einer neuen Menge S'_c gesammelt (beachte hierbei, dass Suffixe von Suffixen von $t\$$ wiederum Suffixe von $t\$$ sind). An dem neu konstruierten Kind wird nun die Prozedur rekursiv mit der Menge S'_c aufgerufen. Siehe hierzu den Algorithmus in Abbildung 2.10.

Da Werte immer nur geschrieben und nie modifiziert werden, erklärt sich nun der Teil write-only aus WOTD.

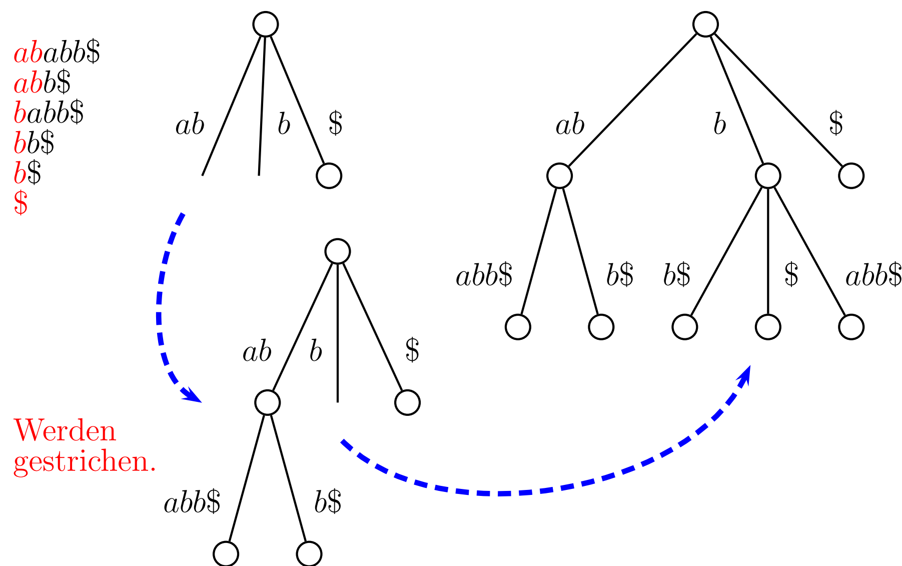


Abbildung 2.11: Beispiel: Konstruktion eines Suffix-Baumes für $t = ababb\$$ mittels WOTD

In der Implementierung wird man die Menge S (bzw. S_c) nicht wirklich als Menge von Wörtern implementieren, da sonst die Menge S bereits quadratischen Platzbedarf hätte. Da S jeweils eine Menge von Suffixen beschreibt, wird man diese Menge als Menge der Anfangspositionen der Suffixe in S realisieren.

Ein Beispiel für den Ablauf dieses Algorithmus mit dem Wort $t = ababb\$$ ist in Abbildung 2.11 angegeben. Wenn man den WOTD-Algorithmus für Worte ohne das Endsymbol \$, verwendet, entstehen nicht notwendigerweise Suffix-Bäume, es kann dann interne Konten mit nur einem Kind geben (betrachte beispielweise das Wort $t = abab$).

2.3.2 Zeitbedarf

Wir wollen nun den Zeitbedarf des Algorithmus analysieren. Für jeden Knoten wird ein Bucket-Sort nach dem ersten Zeichen ausgeführt. Dies geht in Zeit $O(|S|)$. Wir verteilen diese Kosten auf das jeweils erste Zeichen in den Mengen S_c. Somit erhält jedes erste Zeichen in S_c konstante Kosten.

Zur Ermittelung des längsten gemeinsamen Präfixes p in S_c werden in jeder Zeichenreihe in S_c maximal $|p|$ Vergleiche ausgeführt. Das erste Zeichen ist in allen Wörtern in S_c sind nach dem Bucket-Sort gleich (und muss nicht verglichen werden), nach Definition von p muss es an Position $p + 1$ in S_c zwei verschiedene Zeichen geben. Diese Kosten verteilen wir nun auf die Zeichen des längsten gemeinsamen Präfi-

xes. Da nach Vorsortierung $|p| \geq 1$ gilt, erhält auch hier wieder jedes Zeichens des längsten gemeinsamen Präfixes konstante Kosten.

Da anschließend das längste gemeinsame Präfix entfernt wird, enthält jedes Zeichen aus $S(t)$ maximal konstant viele Einheiten. Da für ein Wort t der Länge n gilt, dass die Anzahl der Zeichen aller Suffixe von t gleich $\binom{n+1}{2}$ ist, beträgt die Laufzeit im worst-case $O(n^2)$.

Theorem 2.8 *Für ein Wort $t \in \Sigma^n$ mit $n \in \mathbb{N}$ kann der zugehörige Suffix-Baum mit dem WOTD-Algorithmus im worst-case in Zeit $O(n^2)$ konstruiert werden.*

In der Praxis bricht der Algorithmus ja ab, wenn $|S_c| = 1$ ist. Für jeden internen Knoten v eines Suffix-Baumes gilt im average-case $|\text{path}(v)| = O(\log_{|\Sigma|}(n))$. Somit werden im average-case maximal die ersten $O(\log_{|\Sigma|}(n))$ Zeichen eines Wortes aus $S(t)$ betrachtet. Daraus ergibt sich dann im average-case eine Laufzeit von $O(n \log_{|\Sigma|}(n))$.

Theorem 2.9 *Für ein Wort $t \in \Sigma^n$ mit $n \in \mathbb{N}$ kann der zugehörige Suffix-Baum mit dem WOTD-Algorithmus im average-case in Zeit $O(n \log_{|\Sigma|}(n))$ konstruiert werden.*

2.4 Der Algorithmus von Ukkonen

In diesem Abschnitt wollen wir einen Algorithmus vorstellen, der im worst-case eine lineare Laufzeit zur Konstruktion eines Suffix-Baumes besitzt.

2.4.1 Suffix-Links

Zunächst einmal benötigen wir für die Darstellung des Algorithmus den Begriff eines Suffix-Links.

Definition 2.10 *Sei $t \in \Sigma^*$ und sei T der Suffix-Baum zu t. Zu jedem inneren Knoten $\overline{aw}$ von T mit $a \in \Sigma$, $w \in \Sigma^*$ (also mit Ausnahme der Wurzel) ist der* Suffix-Link *des Knotens $\overline{aw}$ als $\overline{w}$ definiert (in Zeichen slink$(\overline{aw}) = \overline{w}$).*

In Abbildung 2.12 sind die Suffix-Links für den Suffix-Baum für $t = ababb$ als gestrichelte Linien dargestellt.

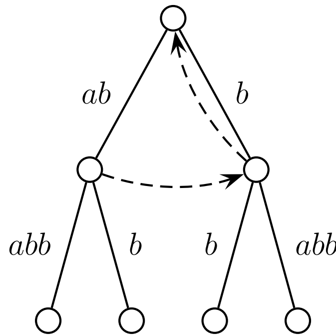


Abbildung 2.12: Beispiel: Die Suffix-Links im Suffix-Baum für $t = ababb$

Als nächstes sollten wir uns noch überlegen, dass Suffix-Links überhaupt wohldefiniert sind. Es könnte ja durchaus sein, dass es einen Knoten $\overline{aw}$ für $a \in \Sigma$ und $w \in \Sigma^*$ gibt, aber es keinen Knoten v mit $\text{path}(v) = w$ gibt.

Wir müssen also Folgendes zeigen: Wenn $\overline{aw}$ ein innerer Knoten des Suffix-Baums ist, dann ist auch $\overline{w}$ ein innerer Knoten. Beachte dabei, dass nach Definition $\overline{aw}$ nicht die Wurzel sein kann.

Ist $\overline{aw}$ ein innerer Knoten des Suffix-Baumes (ungleich der Wurzel), dann muss es Zeichen $b \neq c \in \Sigma$ geben, so dass sowohl $awb \sqsubseteq t$ als auch $awc \sqsubseteq t$ gilt. Dabei sind bx und cx' mit $x, x' \in \Sigma^*$ die Kantenmarkierungen von (mindestens) zwei ausgehenden Kanten aus dem Knoten $\overline{aw}$. Dies folgt aus der Definition des Suffix-Baumes als kompakter Σ^+-Baum und ist in Abbildung 2.13 schematisch dargestellt.

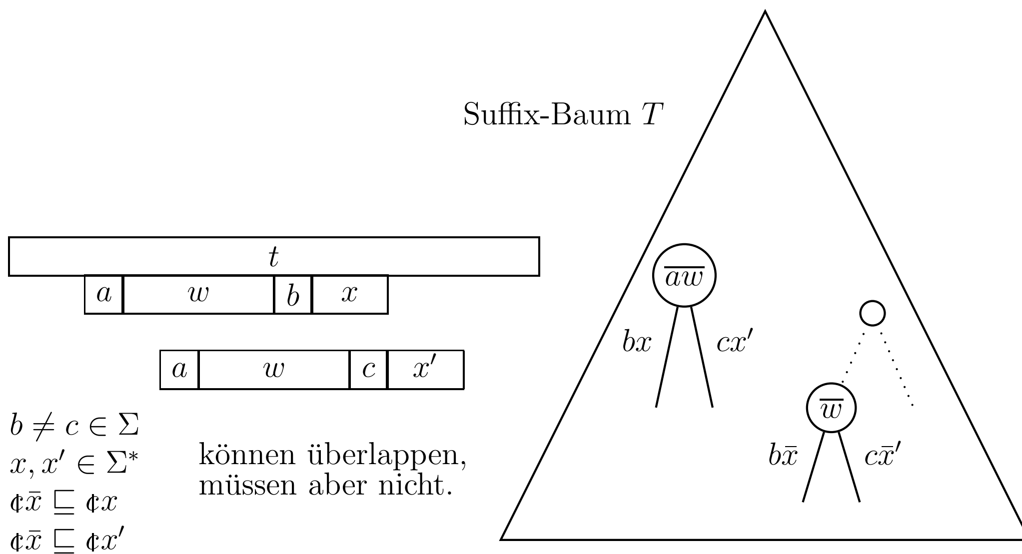


Abbildung 2.13: Schema: Suffix-Links sind wohldefiniert

Gilt jedoch sowohl $awb \sqsubseteq t$ als auch $awc \sqsubseteq t$, dann gilt auch $wb \sqsubseteq t$ und $wc \sqsubseteq t$. Somit muss $\overline{w}$ ein Knoten im Suffix-Baum für t sein.

Man beachte, dass man für Blätter eines Suffix-Baumes auf diese Weise keine Suffix-Links definieren kann. Hier kann es passieren, dass für ein Blatt $\overline{aw}$ kein Knoten $\overline{w}$ im Suffix-Baum existiert. Der Leser möge sich solche Beispiele selbst überlegen.

2.4.2 Verschachtelte Suffixe und verzweigende Teilwörter

Für die folgende Beschreibung des Algorithmus von Ukkonen zur Konstruktion von Suffix-Bäumen benötigen wir noch einige Definitionen und Notationen, die in diesem Abschnitt zur Verfügung gestellt werden.

Definition 2.11 *Sei $t \in \Sigma^*$. Ein Suffix $s \sqsubseteq t$ heißt* verschachtelt *(engl.* nested*), wenn sowohl $s\$ \sqsubseteq t\$$ als auch $sa \sqsubseteq t$ für ein $a \in \Sigma$ gilt.*

In Abbildung 2.14 ist ein verschachteltes Suffix s von t schematisch dargestellt. Man beachte hierbei, dass anders als in der Skizze das verschachtelte Vorkommen mit dem Suffix auch überlappen kann.

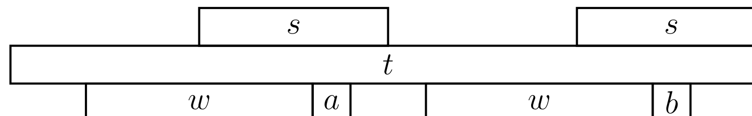


Abbildung 2.14: Skizze: verschachteltes Suffix s und rechtsverzweigendes Teilwort w

Definition 2.12 *Sei $t \in \Sigma^*$. Ein Teilwort $w \sqsubseteq t$ heißt* rechtsverzweigend *(engl.* rightbranching*), wenn es $a \neq b \in \Sigma$ mit $wa \sqsubseteq t$ und $wb \sqsubseteq t$ gibt.*

Ein rechtsverzweigendes Teilwort w von t ist in Abbildung 2.14 schematisch dargestellt Man beachte, dass anders als in der Skizze das Vorkommen des rechtsverzweigenden Teilwortes auch überlappen kann.

Für das Wort $t = abbabbba$ ist beispielsweise ba oder bba ein verschachteltes Suffix und bb ein rechtsverzweigendes Teilwort.

2.4.3 Idee von Ukkonens Algorithmus

Sei $t = t_1 \cdots t_n \in \Sigma^*$. Der Algorithmus konstruiert dann sukzessive Suffix-Bäume $T^1, \ldots, T^n$ mit $T^i = T(t_1 \cdots t_i)$. Nach Definition stellt T^i alle Teilwörter von $t_1 \cdots t_i$ und T^{i+1} alle Teilwörter von $t_1 \cdots t_i \cdot t_{i+1}$ dar, d.h words$(T^i) = \{w : w \sqsubseteq t_1 \cdots t_i\}$ und words$(T^{i+1}) = \{w : w \sqsubseteq t_1 \cdots t_{i+1}\}$.

T^1 ist offensichtlich leicht zu erstellen. Da jedes Teilwort von $t_1 \cdots t_i$ auch ein Teilwort von $t_1 \cdots t_i \cdot t_{i+1}$ ist, ist T^{i+1} quasi eine Erweiterung von T^i. Wir müssen uns nur noch überlegen, wie wir T^{i+1} aus T^i konstruieren können.

Im Weiteren verwenden wir die folgenden Abkürzungen: $x := t_1 \cdots t_i$ und $a := t_{i+1}$. Also müssen wir in T^i alle Teilwörter aus xa einfügen, die keine Teilwörter von x sind, um T^{i+1} zu erhalten. Dazu halten wir erst noch einige Notationen fest.

Notation 2.13 $I := \{w \in \Sigma^* : w \sqsubseteq xa \wedge w \not\sqsubseteq x\}$.

Halten wir zuerst die folgenden beiden offensichtlichen Lemmata fest.

Lemma 2.14 *Alle Wörter aus I sind Suffixe von xa.*

Beweis: Dies folgt unmittelbar aus der Definition von I. ■

Lemma 2.15 *Sei $sa \in I$, dann ist $\overline{sa}$ in T^{i+1} ein Blatt.*

Beweis: Da sa nach dem vorhergehenden Lemma ein Suffix von xa ist und sa kein Teilwort von x ist, folgt die Behauptung. ■

Damit können wir noch folgende wichtige Notation einführen.

Notation 2.16 $I^* := \{sa \in I : \overline{s}$ *ist kein Blatt in* $T^i\} \subseteq I$.

Für alle Wörter in $sa \in I \setminus I^*$ ist nicht sonderlich viel zu tun, da es sich bei $\overline{s}$ um ein Blatt in T^i handelt. Für die Konstruktion von T^{i+1} wird an die Kantenmarkierung des zu $\overline{s}$ inzidenten Blattes nur ein a angehängt und die Bezeichnung dieses Blattes wird zu $\overline{sa}$. Somit sind alle Wörter aus $I \setminus I^*$ nun ebenfalls dargestellt. Die eigentliche Arbeit besteht also nur bei der Darstellung der Wörter aus I^*.

Lemma 2.17 *Für ein Suffix sa von xa gilt genau dann $sa \in I^*$, wenn s ein verschachteltes Suffix von x ist und $sa \not\sqsubseteq x$.*

Beweis: **⇒:** Da $sa \in I^* \subseteq I$ gilt, gilt nach Definition von I, dass $sa \not\sqsubseteq x$.

Da $sa \in I^*$ ist, ist $\overline{s}$ kein Blatt in $T^i = T(x)$. Somit muss s sowohl ein Suffix von x als auch ein echtes Teilwort von x sein, d.h. s ist ein verschachteltes Suffix von x.

⇐: Da s ein verschachteltes Suffix von x ist, ist sa insbesondere ein Suffix von xa. Da außerdem $sa \not\sqsubseteq x$, folgt $sa \in I$.

Da s ein verschachteltes Suffix von x ist, gibt es ein $b \in \Sigma$, so dass auch $sb \sqsubseteq x$ gilt (mit $b \neq a$). Somit kann $\overline{s}$ kein Blatt in $T^i = T(x)$ sein und es folgt $sa \in I^*$. ■

Definition 2.18 *Das längste verschachtelte Suffix von x heißt* aktives Suffix *und wird mit $\alpha(x)$ bezeichnet.*

Als Beispiel hierfür ist das aktive Suffix von $x = ababb$ gleich $\alpha(x) = b$ und von $x = abbab$ gleich $\alpha(x) = ab$. Beachte, dass für das leere Wort kein aktives Suffix definiert ist.

Lemma 2.19 *Sei $x \in \Sigma^+$ und $a \in \Sigma$. Es gelten die folgenden Aussagen:*

1. *Für alle Suffixe s von x gilt: s ist genau dann verschachtelt, wenn $|s| \leq |\alpha(x)|$.*
2. *Für alle Suffixe s von x gilt: $sa \in I^* \Leftrightarrow |\alpha(x)a| \geq |sa| > |\alpha(xa)|$*
3. *$\alpha(xa)$ ist ein Suffix von $\alpha(x) \cdot a$.*
4. *Ist $sa = \alpha(xa)$ und $s \neq \alpha(x)$, dann ist s ein verschachteltes Suffix, sogar ein rechtsverzweigendes Suffix von x.*

Beweis: Zu 1.: **⇒:** Da $\alpha(x)$ das längste verschachtelte Suffix ist, folgt die Behauptung unmittelbar.

⇐: Ist s ein Suffix von x und gilt $|s| \leq |\alpha(x)|$, so ist s ein Suffix von $\alpha(x)$. Da $\alpha(x)$ ein verschachteltes Suffix von x ist, kommt es in x ein weiteres Mal als Teilwort vor und somit auch s. Also ist s ein verschachteltes Suffix. Dies ist in Abbildung 2.15 illustriert.

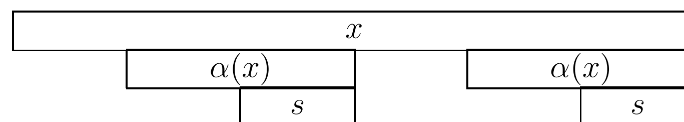


Abbildung 2.15: Skizze: Das Suffix s von x mit $|s| \leq |\alpha(x)|$ ist verschachtelt

Zu 2.: Es gilt offensichtlich mit Lemma 2.17 und mit Teil 1. für jeden Suffix s von x (und somit jeden Suffix sa von xa):

$$\begin{aligned} sa \in I^* \quad &\Leftrightarrow \quad s \text{ ist ein verschachteltes Suffix von } x \text{ und } sa \not\sqsubseteq x \\ &\Leftrightarrow \quad |s| \leq |\alpha(x)| \text{ und } sa \text{ ist kein verschachteltes Suffix von } xa \\ &\Leftrightarrow \quad |s| \leq |\alpha(x)| \wedge |sa| > |\alpha(xa)| \\ &\Leftrightarrow \quad |\alpha(x)a| \geq |sa| > |\alpha(xa)| \end{aligned}$$

Zu 3.: Offensichtlich sind $\alpha(xa)$ und $\alpha(x)a$ Suffixe von xa. Nehmen wir an, dass $\alpha(xa)$ kein Suffix von $\alpha(x) \cdot a$ wäre. Da $\alpha(xa)$ das längste verschachtelte Suffix von xa ist, muss es nochmals als Teilwort in x auftreten. Dann kann aber nicht $\alpha(x)$ das längste verschachtelte Suffix von x sein. Diese Situation ist in Abbildung 2.16 illustriert.

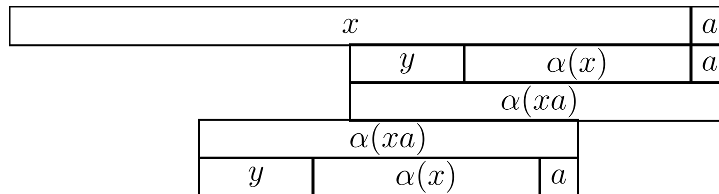


Abbildung 2.16: Skizze: Die Suffixe $\alpha(x) \cdot a$ und $\alpha(xa)$ von xa und ihre Wiederholungen in xa

Zu 4.: Sei also $\alpha(xa) = sa$ und $\alpha(x) \neq s$. Da $\alpha(xa) = sa$ ein verschachteltes Suffix von xa ist, muss auch s ein verschachteltes Suffix von x sein. Somit ist $|\alpha(x)| > |s|$ (da ja nach Voraussetzung $s \neq \alpha(x)$) und s ist ein echtes Suffix von $\alpha(x)$. Nach Definition taucht also $\alpha(x)$ (und damit auch s) ein weiteres Mal in x auf. Da sa ein längstes verschachteltes Suffix von xa ist, muss bei dem weitere Vorkommen von $\alpha(x)$ direkt dahinter ein Zeichen $b \in \Sigma$ mit $b \neq a$ auftauchen (siehe auch die

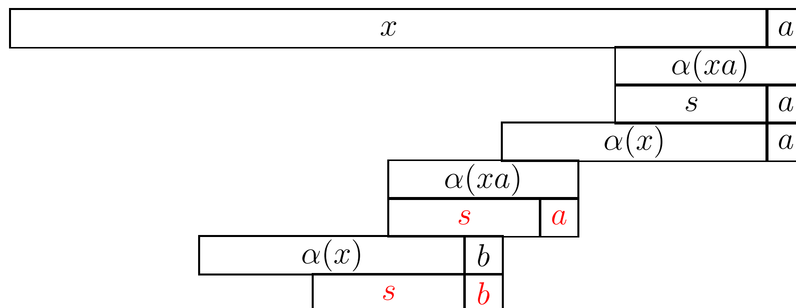


Abbildung 2.17: Skizze: Die Suffixe $\alpha(x) \cdot a$ und $\alpha(xa)$ von xa

Illustration in Abbildung 2.17). Also ist s ein rechtsverzweigendes Suffix von x, da in x sowohl sa als auch sb auftritt.

Somit ist der gesamte Beweis abgeschlossen. ■

19.11.20

2.4.4 Ukkonens Online Algorithmus

Die Aussage 2 des Lemmas 2.19 gibt uns eine Charakterisierung derjenigen Suffixe, die in T^i noch dargestellt werden müssen, um T^{i+1} zu erhalten. Mit Hilfe von Aussage 3 des Lemmas 2.19 wissen wir weiter, dass die noch darzustellenden Suffixe Suffixe des alten aktiven Suffixes verlängert um a sind.

Wie können wir nun alle neu darzustellenden Suffixe von xa finden? Wir beginnen mit dem Suffix $\alpha(x)$ in T^i und versuchen dort das Zeichen a anzuhängen. Anschließend durchlaufen wir alle Suffixe von $\alpha(x)$ in abnehmender Länge (was sich elegant durch die Suffix-Links erledigen lässt) und hängen jeweils ein a an. Wir enden mit dieser Prozedur, wenn für ein Suffix s von $\alpha(x)$ das Wort sa bereits im Suffix-Baum dargestellt ist, d.h. die einzufügende Kante mit Kantenmarkierung a bereits existiert (bzw. eine Kanten, deren Kantenmarkierung nach (einem Suffix von) s bereits ein a enthält). Nach dem vorhergehenden Lemma erhalten wir durch Ablaufen der Kantenmarkierung a den aktiven Suffix $\alpha(xa)$ von T^{i+1}. Diese Position wird dann als Startposition für die Erweiterung auf den Suffix-Baum T^{i+1} verwenden.

Wir müssen jedoch noch berücksichtigen, dass wir auf einen Suffix s von $\alpha(x)$ treffen, wobei $\overline{s}$ kein Knoten in T^i ist. In diesem Fall müssen wir in die Kante, an der die

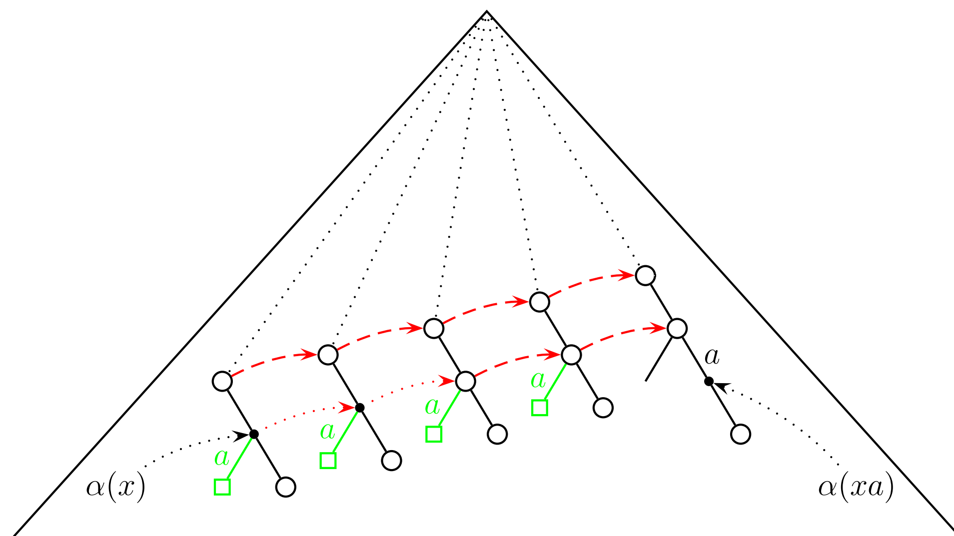


Abbildung 2.18: Skizze: Übergang von T^i zu T^{i+1} in Ukkonens Algorithmus

Darstellung von s endet, aufbrechen und an der geeigneten Stelle einen neuen Knoten einfügen.

Die Erweiterung eines Suffix-Baumes für x zu einem Suffix-Baum für xa ist in Abbildung 2.18 schematisch dargestellt. Dort sind die rot gepunkteten Linien die noch nicht vorhanden Suffix-Links, denen wir gerne folgen möchten, die aber gar nicht existieren, da deren Anfangspunkt in T^i gar nicht zu einem Knoten im Baum T^i gehören (diese werden aber im Laufe der Erweiterung hinzugefügt). Die rot gestrichelten Linien sind wirklich vorhandene Suffix-Links in T^i .

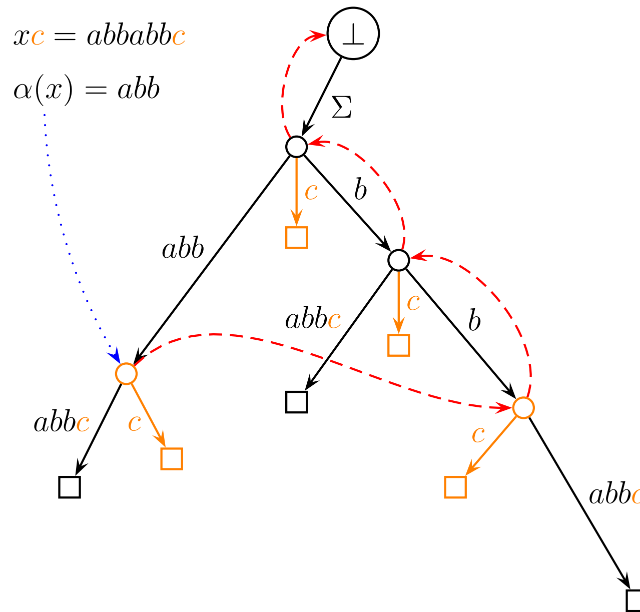


Abbildung 2.19: Beispiel: Konstruktion von $T(abbabbc)$ aus $T(abbabb)$

In Abbildung 2.19 ist ein Beispiel für eine Erweiterung eines Suffix-Baumes angegeben. Dort wird auch das mehrfache Aufbrechen von Kanten illustriert. Die Erweiterung ist orangefarben dargestellt.

In diesem Beispiel haben wir noch eine virtuelle Superwurzel $\perp$ eingeführt. Mit $\text{slink}(\overline{\varepsilon}) = \perp$ können wir sicherstellen, dass jeder innere Knoten einen Suffix-Link besitzt. Dazu nehmen wir noch eine Baum-Kante von $\perp$ zu $\overline{\varepsilon}$ an, die mit allen Zeichen aus Σ markiert ist. Damit stellen wir sicher, dass wir wieder zur Wurzel zurückkehren können, wenn wir eigentlich fälschlicherweise dem Suffix-Link von $\overline{\varepsilon}$ zu $\perp$ gefolgt sind.

Wie wir noch sehen werden, ist der einzige Grund für die Einführung von $\perp$, dass die Beschreibung von Ukkonens Algorithmus einheitlicher wird, da wir beim Folgen der Suffix-Links dann immer irgendwann auf einen Knoten stoßen werden, von dem eine Kante mit dem Zeichen $a \in \Sigma$ ausgeht.

```
Ukkonen (string t = t_1 ··· t_n)
begin
    tree T := T(t_1);
    string α(t_1) := ε;
    for (i := 1; i < n; i++) do
        string x := t_1 ··· t_i;
        char a := t_{i+1};
        string s := α(x);
        while (sa is not represented in T) do
            insert \overline{sa} in T;
            s := s_2 ··· s_{|s|};                         /* via suffix-links */
                                   /* Note that s_2 ··· s_{|s|} := ⊥ iff s = ε */
        α(xa) := sa ;             /* Note that ⊥ · a := ε for any a ∈ Σ! */
end
```

Abbildung 2.20: Algorithmus: Abstrakte Fassung von Ukkonens Algorithmus

Wie finden wir in diesem Beispiel den Suffix-Link von $\alpha(x) = abb$? Im Suffix-Baum für *abbabb* gibt es ja keinen Knoten $\overline{abb}$. Wir folgen stattdessen dem Suffix-Link des Knotens, der als erstes oberhalb liegt: $\overline{\varepsilon}$. Somit landen wir mit dem Suffix-Link im Knoten $\perp$. Vom Knoten $\overline{\varepsilon}$ mussten wir ja noch die Zeichenreihe *abb* ablesen um bei der Position von $\alpha(x)$ zu landen. Dies müssen wir jetzt auch vom Knoten $\perp$ aus tun. Damit landen wir dann eigentlich beim Knoten $\overline{bb}$, der aber dummerweise im Suffix-Baum für *abbabb* auch nicht existiert (der aber jetzt eingefügt wird).

Für das weitere Folgen des Suffix-Links von $\overline{bb}$ (wobei es den Suffix-Link ja noch nicht gibt) starten wir wieder von $\overline{b}$ aus. Nach Folgen des Suffix-Links landen wir in $\overline{\varepsilon}$. Nun laufen wir noch das Wort b ab, um die eigentlich Position $\overline{b}$ im Suffix-Baum zu finden, an der wir weitermachen. Die folgenden Schritte sind nun einfach und bleiben dem Leser zur Übung überlassen.

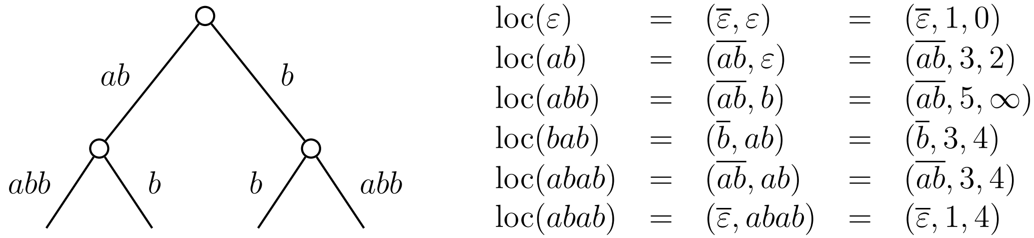
Die sich aus dieser Diskussion ergebende, erste abstrakte Fassung von Ukkonens Algorithmus ist in Abbildung 2.20 angegeben.

Um dies etwas algorithmischer beschreiben zu können, benötigen wir erst noch eine angemessene Darstellung der vom Suffix-Baum T dargestellten Wörter, die wir als Lokation bezeichnen wollen (in Zeichen $\text{loc}(v)$).

Definition 2.20 *Sei $t \in \Sigma^*$ und sei $T = T(t)$ der zugehörige Suffix-Baum. Für $s \in words(T)$ ist* $\text{loc}(s) = (\overline{u}, v) \equiv (\overline{u}, j + |u|, j + |s| - 1)$ *eine* Lokation *von s in T, wenn $\overline{u} \in V(T)$, $s = uv$ für ein $v \in \Sigma^*$ und $t_j \cdots t_{j+|s|-1} = s$ gilt. Ist $\overline{s}$ ein Blatt in T, dann muss darüber hinaus $u \neq s$ gelten*

Für eine Lokation $(\overline{v}, i, j)$ eines Blattes schreiben wir darüber hinaus im Folgenden $(\overline{v}, i, \infty)$, da wir für ein Blatt immer $j = |t|$ wählen können. Diese Schreibweise bezeichnet man als *offene Lokation* bzw. auch als *offene Referenz*, da der Endpunkt keine Rolle spielt. Das ist auch ein Grund, warum Lokationen keine Blätter verwenden dürfen.

Die Definition wollen wir uns in Abbildung 2.21 noch einmal an einem Beispiel genauer anschauen.



$$
\begin{array}{lclcl}
\mathrm{loc}(\varepsilon) & = & (\overline{\varepsilon}, \varepsilon) & = & (\overline{\varepsilon}, 1, 0) \\
\mathrm{loc}(ab) & = & (\overline{ab}, \varepsilon) & = & (\overline{ab}, 3, 2) \\
\mathrm{loc}(abb) & = & (\overline{ab}, b) & = & (\overline{ab}, 5, \infty) \\
\mathrm{loc}(bab) & = & (\overline{b}, ab) & = & (\overline{b}, 3, 4) \\
\mathrm{loc}(abab) & = & (\overline{ab}, ab) & = & (\overline{ab}, 3, 4) \\
\mathrm{loc}(abab) & = & (\overline{\varepsilon}, abab) & = & (\overline{\varepsilon}, 1, 4)
\end{array}
$$

Abbildung 2.21: Beispiel: Einige Lokationen im Suffix-Baum für *ababb*

Für unseren Algorithmus benötigen wir insbesondere so genannte kanonische Lokationen.

Definition 2.21 *Eine Lokation* $\mathrm{loc}(s) = (\overline{v}, w) \equiv (\overline{v}, i, j)$ *heißt* kanonisch, *wenn für jede andere Lokation* $\mathrm{loc}(s) = (\overline{v'}, w') \equiv (\overline{v'}, i', j')$ *gilt, dass* $|v| > |v'|$.

Für die in Abbildung 2.21 angegebenen Lokationen sind die ersten fünf kanonische, die sechste jedoch nicht.

Für eine kanonische Lokation gilt also das Folgende:

- Wenn $\overline{s}$ ein innere Knoten (Verzweigungsknoten) in T ist, dann gilt
  $$\mathrm{loc}(s) := (\overline{s}, \varepsilon) \equiv (\overline{s}, j + |s|, j + |s| - 1),$$
  wobei $t_j \cdots t_{j+|s|-1} = s$.
- Wenn $\overline{s}$ ein Blatt ist, d.h. $\overline{u} \stackrel{v}{\to} \overline{s}$ und $s = uv$, dann gilt
  $$\mathrm{loc}(s) := (\overline{u}, v) \equiv (\overline{u}, j + |u|, \infty),$$
  wobei $t_j \cdots t_{j+|uv|-1} = uv = s$.
- Wenn kein Knoten $\overline{s}$ in T existiert, dann existiert eine Kante $\overline{u} \stackrel{vw}{\to} \overline{uvw}$ mit $s = uv$, $v \neq \varepsilon$, $w \neq \varepsilon$ und es gilt
  $$\mathrm{loc}(s) := (\overline{u}, v) \equiv (\overline{u}, j + |u|, j + |s| - 1),$$
  wobei $t_j \cdots t_{j+|uv|-1} = uv = s$.

```
Ukkonen (string t = t_1 ··· t_n)
begin
    tree T := T(t_1);
    ref (v̄, w) := (r(T), ε);                          /* r(T) is the root of T */
    for (i := 1; i < n; i++) do
        node x := y := NIL;
        while (not T.lookup((v̄, w), t_{i+1})) do /* i.e., while((v̄, w · t_{i+1}) ∉ T) */
            y := T.insert((v̄, w), t_{i+1}) ;          /* returns parent of new leaf */
            if (x ≠ NIL) then
                slink(x) := y;
            x := y;
            (v̄, w) := canonize((slink(v̄), w));
        if (x ≠ NIL) then
            slink(x) := v̄;                    /* Note that always w = ε if x ≠ NIL */
        (v̄, w) := canonize((v̄, w · t_{i+1}));
end
```

Abbildung 2.22: Algorithmus: Ukkonens Algorithmus

Aus unserer Diskussion ergibt sich nun Ukkonens Algorithmus, der im Pseudo-Code in Abbildung 2.22 angegeben ist. In der Regel wird dabei in einer realen Implementierung allerdings w dabei als Referenz (j, k) angegeben, d.h. $w = t_j \cdots t_k$. Wir gehen darauf im Folgenden allerdings nicht immer im Detail ein, siehe aber auch Abbildung 2.23. Bei dieser Realisierung wird die Referenz (j, k) allerdings immer auf einen Suffix von $t_1 \cdots t_i$ verweisen, d.h. es gilt $k = i$.

Hierbei schaut die Prozedur lookup$((\overline{v}, w), a)$ nach, ob im Baum T das Wort vwa dargestellt wird, d.h. ob man ab der kanonischen Lokation $(\overline{v}, w)$ in T den Buchstaben a weiterverfolgen kann. Ist $w = \varepsilon$, dann wird nur überprüft, ob $\overline{v}$ eine ausgehende Kante hat deren Kantenmarkierung mit a beginnt. Ist $w \neq \varepsilon$, dann wird die ausgehende Kante von $\overline{v}$ betrachtet, deren Kantenmarkierung mit dem ersten Buchstaben von w beginnt. In dieser Kantenmarkierung wird jetzt nur der Buchstabe an Position $|w| + 1$ mit a verglichen. Dabei liefert lookup `false`, wenn $(\overline{v}, w \cdot a)$ keine Lokation in T ist, andernfalls wird `true` zurückgeliefert. Der Leser möge sich selbst davon überzeugen, dass in dem im Algorithmus benötigten Fall (nach dem Ende der while-Schleife) der Knoten $\overline{vw}$ in T wirklich existieren muss.

Man beachte, dass die Buchstaben an den Positionen 2 mit $|w|$ nicht mit der entsprechenden Kantenmarkierung verglichen werden müssen, da diese nach Konstruktion übereinstimmen müssen. Dies folgt daraus, dass das zur Lokation $(\overline{v}, w)$ gehörige Wort vw nach Konstruktion von Ukkonens Algorithmus in T zwingend dargestellt sein muss.

Für jeden Aufruf lookup$((\overline{v}, w), t_i)$ (also lookup$((\overline{v}, j, k), t_i)$) bedeutet dies, dass man für $w = \varepsilon$ (also $j > k$) eine ausgehende Kanten von $\overline{v}$ finden muss, deren Kantenmarkierung mit t_i beginnt. Andernfalls muss man zuerst eine ausgehende Kanten von $\overline{v}$ finden, deren Kantenmarkierung (p, q) mit t_j beginnt, d.h. ob $t_j = t_p$; falls ja, muss noch die Bedingung $t_i = t_{p+k-j+1}$ geprüft werden. Falls einer der Fälle eintritt, liefert lookup den Wert `true`, sonst `false`.

Die Prozedur canonize macht aus einer gegebenen Lokation $(\overline{v}, w)$ eine kanonische, indem sie vom Knoten $\overline{v}$ soweit im Suffix-Baum die Zeichenreihe w verfolgt, bis die kanonische Lokation gefunden wird. Man beachte da, dass der Aufwand proportional zur Anzahl überlaufener Kanten plus 1 ist, da jeweils nur das erste Zeichen einer ausgehenden Kante betrachtet werden muss (die folgenden Zeichen dieser Kantenmarkierung müssen dann, wie schon diskutiert, auch wieder alle übereinstimmen).

Wie man in Abbildung 2.19 sehen kann, kann es auch wirklich passieren, dass man bei canonize mehrere Kanten überlaufen muss. Dort ist die kanonische Lokation des Suffixes $\alpha(x) = abb$ gerade $(\overline{\varepsilon}, abb)$. Folgt man dem zugehörigen Suffix-Link, so gelangt man zur Lokation $(\perp, abb)$. Um die zugehörige kanonische Lokation $(\overline{b}, b)$ zu erhalten, muss man zwei Knoten für das Wort abb im Baum hinabsteigen.

Innerhalb von canonize muss für eine Lokation $(\overline{v}, w)$ (also $(\overline{v}, j, k)$) zuerst die von $\overline{v}$ ausgehende Kante mit Kantenmarkierung (p, q) zu $\overline{vw'}$ bestimmt werden, die mit t_j beginnt, d.h. $t_j = t_p$. Ist $|w'| > |w|$ (also $q - p > k - j$) oder ist $|w'| = |w|$ (also $q - p = k - j$) und $\overline{vw}$ ein Blatt, dann ist die Lokation kanonisch. Ansonsten wird in Canonize versucht die Lokation $(\overline{vw'}, w'')$ mit $w = w'w''$ (also $(\overline{vw'}, j + (q - p + 1), k)$) zu kanonisieren.

Die Prozedur insert$((\overline{v}, w), a)$ fügt an der Lokation $(\overline{v}, w)$ eine Kante mit Kantenmarkierung a zu einem neuen Blatt ein. War $(\overline{v}, w)$ bereits ein Knoten im Suffix-Baum, so wird die neue Kante dort nur angehängt. Beschreibt die Lokation $(\overline{v}, w)$ eine Position innerhalb einer Kante, so wird in diese Kante an der zur Lokation $(\overline{v}, w)$ entsprechenden Stelle ein neuer Knoten eingefügt, an den die Kante zu dem neuen Blatt angehängt wird. Man überlegt sich leicht, dass dies in konstanter Zeit erledigt werden kann. Der zu dem neuen Blatt adjazente innere Knoten kann dann leicht wie gefordert zurückgeliefert werden.

Dabei muss auch noch ein Suffix-Link vom vorhergehenden Knoten auf diesen neu eingefügten Knoten gesetzt werden. Mit einer geeigneten Buchhaltung kann auch dies in konstanter Zeit erledigt werden, da der Knoten, der den neuen Suffix-Link auf den neu eingefügten bzw. aktuell betrachteten Knoten erhält, gerade eben vorher betrachtet wurde.

Für jeden Aufruf insert$((\overline{v}, w), t_i)$ (also insert$((\overline{v}, j, k), t_i)$) wird im Falle $w = \varepsilon$ (also $j > k$) nur ein Blatt mit Kantenmarkierung $(i + 1, \infty)$ an den Knoten $\overline{v}$ gehängt.

```
Ukkonen (string t = t_1 ··· t_n)
begin
    tree T := T(t_1);
    ref (v̄, j, k) := (r(T), 2, 1);                      /* Note that always k = i */
    for (i := 1; i < n; i++) do
        node x := y := NIL;
        while (not T.lookup((v̄, j, k), t_{i+1})) do      /* while((v̄, j, k + 1) ∉ T) */
            y := T.insert((v̄, j, k), t_{i+1}) ;          /* returns parent of new leaf */
            if (x ≠ NIL) then
                slink(x) := y;
            x := y;
            (v̄, j, k) := canonize((slink(v̄), j, k));
        if (x ≠ NIL) then
            slink(x) := v̄;                               /* Note that always j > k if x ≠ NIL */
        (v̄, j, k) := canonize(v̄, j, k + 1);
end
```

Abbildung 2.23: Algorithmus: Ukkonens Algorithmus mit Referenzen

Andernfalls muss die Kante mit Kantenmarkierung (p, q) vom Knoten $\overline{v}$ zum Knoten z mit $t_p = t_j$ aufgebrochen werden. Es wird ein neuer Knoten y eingefügt, wobei y den Knoten z als das Kind von $\overline{v}$ ersetzt und die Kantemarkierung $(p, p + k - j)$ erhält. z wird dann zu einem Kind von y mit Kantenmarkierung $(p + k - j + 1, q)$. Weiter bekommt y ein neues Blatt als Kind mit Kantenmarkierung $(i + 1, \infty)$.

Abschließend geben wir noch die Fassung von Ukkonens Algorithmus mit Referenzen in Abbildung 2.23 an.

2.4.5 Zeitanalyse

Es bleibt noch die Analyse der Zeitkomplexität von Ukkonens Algorithmus. Das Einfügen eines neuen Knotens kann, wie bereits diskutiert, in konstanter Zeit geschehen. Da ein Suffix-Baum für t maximal $O(|t|)$ Knoten besitzt, ist der Gesamtzeitbedarf hierfür $O(|t|)$.

Der Hauptaufwand liegt in der Kanonisierung der Lokationen. Für einen Knoten kann eine Kanonisierung nämlich mehr als konstante Zeit kosten. Bei der Kanonisierung wird jedoch die Lokation $(\overline{v}, w)$ zu $(\overline{vv'}, w')$ für $v' \in \Sigma^+$ und $w' \in \Sigma^*$ mit $v'w' = w$. Damit wandert das Ende der Zeichenreihe v des in der Lokation verwendeten internen Knotens bei einer Kanonisierung immer weiter zum Wortende von t hin. Zwar wird $\overline{v}$ beim Ablaufen der Suffix-Links verkürzt, aber dies geschieht nur

am vorderen Ende. Somit kann eine Verlängerung am Wortende von v bei der verwendeten Lokation $(\overline{v}, w)$ maximal $|t|$ Mal auftreten. Also haben alle Aufrufe zur Kanonisierung eine lineare Laufzeit.

Theorem 2.22 *Ein Suffix-Baum für ein Wort $t = t_1 \cdots t_n \in \Sigma^n$ kann mit Ukkonens Algorithmus in Zeit $O(n)$ mit Platzbedarf $O(n)$ konstruiert werden.*

Wir erwähnen hier noch, dass der Algorithmus folgende Eigenschaft einer Darstellung eines Suffix-Baumes benötigt: Man muss von einem Knoten das Kind finden können, das sich über diejenige Kanten erreichen lässt, dessen erstes Zeichen der Kantenmarkierung vorgegeben ist. Verwendet man also für die Repräsentation aus Platzgründen keine Felder, so erhöht sich der Aufwand in der Regel um den Faktor $|\Sigma|$, was bei kleinen Alphabeten noch tolerierbar ist.

Des Weiteren wollen wir noch anmerken, dass in manchen Lehrbüchern eine Laufzeit für Ukkonens Algorithmus von $O(n \log(n))$ angegeben wird. Dabei wird jedoch statt des uniformen Kostenmaßes das logarithmische Kostenmaß verwendet (also die Bit-Komplexität). In diesem Fall hat jedoch der Suffix-Baum selbst schon die Größe $O(n \log(n))$, da dort ja Zahlen (Referenzen und Zeiger) aus dem Intervall $[0 : n]$ vorkommen, deren Darstellung ja $\Theta(\log(n))$ Bits benötigt. Daher handelt es sich auch in diesem Modell um einen asymptotisch optimalen Algorithmus zur Erstellung von Suffix-Bäumen, da die Ausgabegröße bereits $O(n \log(n))$ betragen kann. Man beachte jedoch, dass die Eingabegröße nur $\Theta(n \log(|\Sigma|))$ beträgt. Im logarithmischen Kostenmaß ist also der Suffix-Baum im Allgemeinen größer als die Eingabe!

26.11.20

Repeats 3

3.1 Exakte und maximale Repeats

In diesem Kapitel wollen wir uns mit Wiederholungen (so genannten) Repeats in Zeichenreihen (insbesondere in Genomen) beschäftigen. Dazu müssen wir uns vor allem überlegen, wie man interessante Wiederholungen charakterisiert.

3.1.1 Erkennung exakter Repeats

Zunächst einmal müssen wir formal definieren, was wir unter einer Wiederholung verstehen wollen. Dazu benötigen wir erst noch die Begriffe einer Hamming- und Alignment-Distanz.

Definition 3.1 *Seien $s, t \in \Sigma^n$, dann ist die* Hamming-Distanz *von s und t definiert als $\delta_H(s,t) := |\{i \in [1:n] \,:\, s_i \neq t_i\}|$.*

Definition 3.2 *Seien $s, t \in \Sigma^*$, dann ist die* Alignment-Distanz *von s und t definiert als $\delta_A(s,t) := \min\left\{d(\hat{s},\hat{t}) \,:\, (\hat{s},\hat{t}) \in \mathcal{A}(s,t)\right\}$ wobei $\mathcal{A}(s,t)$ die Menge der Alignments von s und t und $d(\hat{s},\hat{t})$ ein Distanzmaß für ein Alignment ist.*

Als Alignment-Distanz kann beispielsweise die Anzahl von Substitutionen, Insertionen und Deletionen im Alignment $(\hat{s},\hat{t})$ (also die EDIT-Distanz) verwendet werden. Nun können wir definieren, was eine Wiederholung sein soll.

Definition 3.3 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ eine Zeichenreihe. Ein Paar $((i_1, j_1), (i_2, j_2))$ mit $(i_1, j_1) \neq (i_2, j_2)$ heißt*

- exaktes Paar*, wenn $t_{i_1} \cdots t_{j_1} = t_{i_2} \cdots t_{j_2}$. Das Wort $t_{i_1} \cdots t_{j_1}$ wird dann auch als* exaktes Repeat *bezeichnet. Mit $\mathcal{R}(t)$ bezeichnen wir die Menge aller exakten Paare und mit $\overline{\mathcal{R}}(t)$ die Menge aller exakten Repeats.*
- k-mismatch Repeat*, wenn $\delta_H(t_{i_1} \cdots t_{j_1}, t_{i_2} \cdots t_{j_2}) \leq k$. Mit $\mathcal{R}_k^H(t)$ bezeichnen wir die Menge aller k-mismatch Repeats.*
- k-difference Repeat*, wenn $\delta_A(t_{i_1} \cdots t_{j_1}, t_{i_2} \cdots t_{j_2}) \leq k$. Mit $\mathcal{R}_k^A(t)$ bezeichnen wir die Menge aller k-difference Repeats.*

In Abbildung 3.1 sind schematisch zwei solcher Repeats dargestellt. Man beachte dabei, dass nach Definition ein Repeat sich mit sich selbst überlappen kann (wie bei den roten Teilwörtern) oder es auch mehr als zwei Vorkommen eines Repeats geben kann.

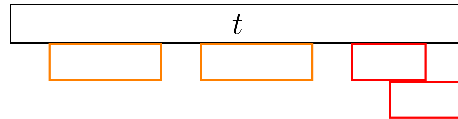


Abbildung 3.1: Skizze: Schematische Darstellung von Repeats

Wir behaupten nun, dass wir alle solchen exakten Repeats in „linearer Zeit" finden können. Dies folgt aus der Tatsache, dass alle exakten Repeats von t zu Lokationen eines Suffix-Baums $T(t\$)$ korrespondieren, die sich nicht auf einer zu einem Blatt inzidenten Kante befinden. Sobald man sich im Suffix-Baum nicht auf einer zu einem Blatt inzidenten Kanten befindet, gibt es Verlängerungsmöglichkeiten, so dass man in mindestens zwei *verschiedenen* Blättern landen kann. Also ist das betrachtete Wort jeweils ein Präfix von mindestens zwei verschiedenen Suffixen, d.h. das betrachtete Wort beginnt an mindestens zwei verschiedenen Positionen in t.

In Abbildung 3.2 ist für das Wort *abbbab* der Suffix-Baum für *abbbab*\$ angegeben und einige darin enthaltene exakte Repeats durch Angabe von exakten Paaren.

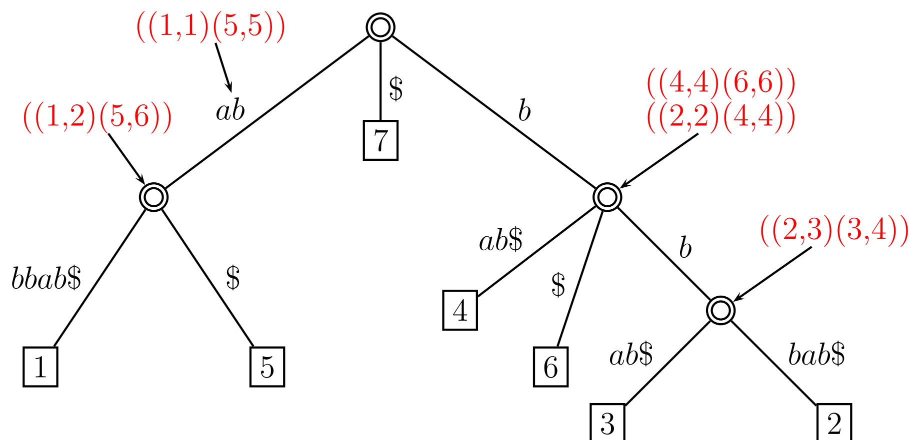


Abbildung 3.2: Beispiel: Suffix-Baum für $t\$ = abbbab\$$ und die exakten Repeats

Durchlaufen wir nun den Suffix-Baum $T(t\$)$ gekürzt um alle Blätter und die dazu inzidenten Kanten mit einer Tiefensuche, so können wir alle dort zu den noch enthaltenen Lokationen korrespondierenden Zeichenreihen ausgeben, und geben damit alle exakten Repeats aus.

Theorem 3.4 *Sei $t \in \Sigma^n$, dann lassen sich alle exakten Repeats als Wörter bzw. Referenzen in Zeit $O(n + k)$ ermitteln, wobei k die Anzahl aller Zeichen in den exakten Repeats bzw. die Anzahl der exakten Repeats ist.*

Somit ist hier mit linearer Zeit gemeint, dass der Algorithmus linear in der Eingabe- und Ausgabegröße läuft. Man beachte, dass es durchaus Wörter über Σ der Länge n geben kann, die $\Theta(n^2)$ exakte Repeats besitzen. Wir merken noch an, dass dieser Algorithmus optimal ist.

Somit haben wir aber nur die exakten Repeats ausgegeben, wir wissen jedoch nicht wo diese auftreten. Daher ist in der Regel ein Algorithmus zum Auffinden aller exakten Paare interessanter. Auch diesen können wir analog zu denen der exakten Repeats definieren. Der Algorithmus zum Auffinden aller exakten Paare geht dabei wie folgt vor:

1. Konstruktion von $T(t\$)$ in Zeit $O(n)$.
2. Tiefensuche durch $T(t\$)$ in Zeit $O(n)$. Während der Tiefensuche führe folgende Schritte aus:
   (a) Jedes Blatt liefert die Indexposition zurück, an der das aufgefundene Suffix beginnt. Der Zeitbedarf hierfür ist insgesamt $O(n)$.
   (b) Jeder innere Knoten liefert eine Liste der Blätter, die von diesem Knoten erreichbar sind, an seinen Elter zurück. Der Zeitbedarf hierfür ist insgesamt $O(n)$, da zum einen immer nur ein Zeiger auf den Beginn und das Ende der Liste übergeben wird und zum anderen, da das Zusammenhängen der Listen der Kinder insgesamt in Zeit $O(n)$ zu realisieren ist (wenn man sich auch jeweils das Ende der Liste merkt).
   (c) Für einen inneren Knoten $\overline{v}$ mit der Blattliste L und mit Elter $\overline{w}$ generiere die exakten Paare $((i, i + \ell - 1), (j, j + \ell - 1))$ mit $i < j \in L$ und $\ell \in [|w| + 1 : |v|]$. Dies lässt sich insgesamt in Zeit $O(n + |\text{output}|)$ erledigen. Nach der vorherigen Diskussion beschreibt jedes ausgegebene exakte Paar einen exakten Repeat. Man überlegt sich leicht, dass alle ausgegeben exakten Paare paarweise verschieden sind.

Theorem 3.5 *Sei $t \in \Sigma^n$, dann lassen sich alle exakten Paare in Zeit $O(n + k)$ ermitteln, wobei k die Anzahl der exakten Paare ist.*

Wir merken noch an, dass dieser Algorithmus optimal ist.

Betrachten wir das Wort $a^n \in \Sigma^*$, dann enthält es nach unserer Definition $\Theta(n^3)$ exakte Paare. Es gilt nämlich, dass für alle $\ell \in [1 : n - 1]$ und $i_1 < i_2 \in [1 : n - \ell + 1]$

das Paar $((i_1, i_1+\ell-1), (i_2, i_2+\ell-1))$ ein exaktes Paar ist. Andererseits existieren nur $n-1$ exakte Repeats a^i in a^n mit $i \in [1:n-1]$.

Somit ist auch hier mit linearer Zeit gemeint, dass der Algorithmus linear in der Eingabe- und Ausgabegröße läuft.

Man kann diesen Algorithmus auch leicht so modifizieren, dass er nur exakte Paare ausgibt, die einen Repeat mit einer Mindestlänge ausgibt. Man muss dann nur innere Knoten berücksichtigen, deren Worttiefe hinreichend groß ist.

3.1.2 Charakterisierung maximaler Repeats

Wie wir gesehen haben, gibt es für die Bestimmung exakter Repeats bzw. Paare einen optimalen Algorithmus. Dennoch muss dieser nicht sehr effizient sein, insbesondere dann nicht, wenn viele exakte Repeats bzw. Paare vorkommen. Daher werden wir die Problemstellung überarbeiten und versuchen uns auf die interessanten Repeats zu beschränken.

Definition 3.6 *Sei* $t = t_1 \cdots t_n \in \Sigma^*$ *und sei* $t' = t'_0 \cdots t'_{n+1} = ¢t_1 \cdots t_n\$$ *mit* $¢ \neq \$ \notin \Sigma$. *Ein Tripel* (i, j, ℓ) *mit* $\ell \in [1:n]$ *sowie* $i < j \in [1:n-\ell+1]$ *heißt* maximales Paar, *wenn* $t_i \cdots t_{i+\ell-1} = t_j \cdots t_{j+\ell-1}$ *und* $t'_{i-1} \neq t'_{j-1}$ *sowie* $t'_{i+\ell} \neq t'_{j+\ell}$ *gilt. Ist* (i, j, ℓ) *ein maximales Paar, dann ist* $t_i \cdots t_{i+\ell-1}$ *ein* maximaler Repeat.

Mit $\mathcal{R}_{\max}(t)$ *bezeichnet man die* Menge aller maximalen Paare *von* t *und mit* $\overline{\mathcal{R}}_{\max}(t) = \{t_i \cdots t_{i+\ell-1} : (i, j, \ell) \in \mathcal{R}_{\max}(t)\}$ *die* Menge aller maximaler Repeats *von* t.

Anschaulich bedeutet dies, dass ein Repeat maximal ist, wenn jede Verlängerung dieses Repeats um ein Zeichen kein Repeat mehr ist. Es kann jedoch Verlängerungen geben, die wiederum ein maximales Repeat sind. Für $t = a\boxed{aa}bb\boxed{aa}ab$ ist aa mit dem maximalen Paar $(2, 6, 2)$ ein maximales Repeat, aber auch $aaab$ ist ein maximales Repeat. Des Weiteren ist für aa das Paar $(2, 7, 2)$ nicht maximal.

Lemma 3.7 *Sei* $t \in \Sigma^*$ *und sei* $T = T(¢t\$)$ *der Suffix-Baum für* $¢t\$$ *mit* $¢, \$ \notin \Sigma$. *Wenn* $\alpha \in \overline{\mathcal{R}}_{\max}(t)$, *dann existiert ein innerer Knoten* v *in* T *mit* $path(v) = \alpha$.

Beweis: Betrachten wir das zu α gehörige maximale Paar (i, j, ℓ). Nach Definition des maximalen Paares gilt insbesondere $t'_{i+\ell} \neq t'_{j+\ell}$. Somit muss also α ein rechtsverzweigendes Teilwort von t' und daher muss $\overline{\alpha}$ ein innerer Knoten in T sein. ■

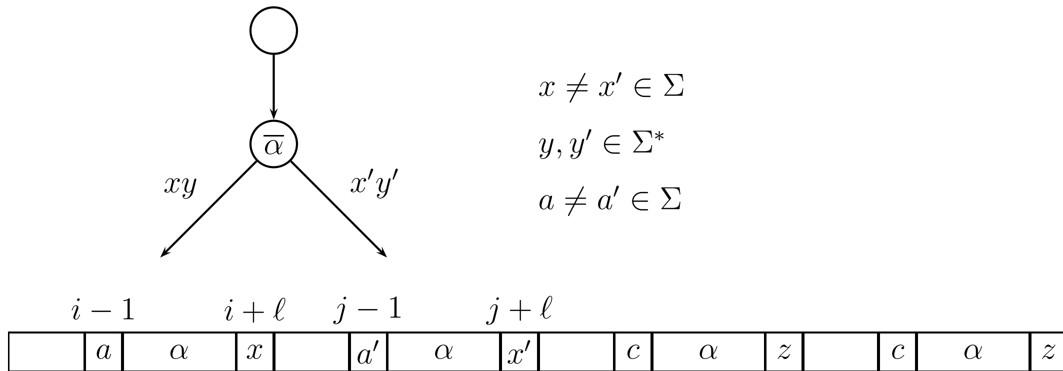


Abbildung 3.3: Skizze: Maximale Repeats enden an inneren Knoten

Die Beweisidee des Lemmas ist noch einmal in Abbildung 3.3 illustriert. Man sollte sich auch klar machen, dass es für ein maximales Repeat α zwei Vorkommen in t existieren müssen, an denen sich die angrenzenden Zeichen unterscheiden (d.h. $t'_{i-1} \neq t'_{j-1}$ und $t'_{i+\ell} \neq t'_{j+\ell}$, wenn das maximale Paar (i, j, ℓ) das maximale Repeat α beschreibt). Es kann durchaus zwei Vorkommen von α in t geben, so dass sich die Wörter nach vorne oder hinten zu längeren Repeats verlängern lassen.

Wir erhalten damit unmittelbar noch das folgende Korollar.

Korollar 3.8 *Ein Wort $t \in \Sigma^*$ besitzt höchstens $|t|$ viele maximale Repeats.*

Beweis: Jedem maximalen Repeat entspricht nach dem vorhergehenden Lemma ein innerer Knoten und in einem Suffix-Baum für ¢t\$ mit $|t| + 2$ Blättern kann es maximal $|t| + 1$ viele innere Knoten geben. In der obigen Formel kann nur dann das Maximum angenommen werden, wenn jeder innere Knoten genau zwei Kinder hat. Da die Wurzel jedoch mindestens drei Kinder hat (über die Kanten, die mit ¢, \$ und einem $a \in \Sigma$ beginnen), kann es maximal $|t|$ innere Knoten geben. ■

Wir merken noch an, dass die Anzahl der maximalen Paare keineswegs linear in der Länge des Textes t begrenzt sein muss. Dies sei dem Leser zur Übung überlassen.

Wie man im Beweis von Lemma 3.7 bemerkt, nutzen wir hier nur aus, dass nach einem maximalen Repeat eines maximalen Paares (i, j, ℓ) die Zeichen unterschiedlich sind (d.h. $t'_{i+\ell} \neq t'_{j+\ell}$), aber nicht die Zeichen unmittelbar davor (d.h. $t'_{i-1} \neq t'_{j-1}$). Um diese bei der Bestimmung maximaler Repeats auch noch berücksichtigen zu können, benötigen wir noch die folgende Definition.

Definition 3.9 *Sei $t \in \Sigma^n$ und sei $t' = t'_0 \cdots t'_{n+1} = ¢t\$$.*

- *Das* Linkszeichen *von Position $i \in [1 : n]$ ist definiert als t'_{i-1}.*
- *Das* Linkszeichen *eines Blattes $\overline{s} \neq \overline{¢t\$}$ von $T(¢t\$)$ ist das Zeichen $t'_{n-|s|+1}$.*
- *Ein innerer Knoten von $T(¢t\$)$ heißt* linksdivers*, wenn in seinem Teilbaum zwei Blätter mit einem verschiedenen Linkszeichen existieren.*

In Abbildung 3.4 ist noch einmal die Definition der Linkszeichen von Blättern illustriert.

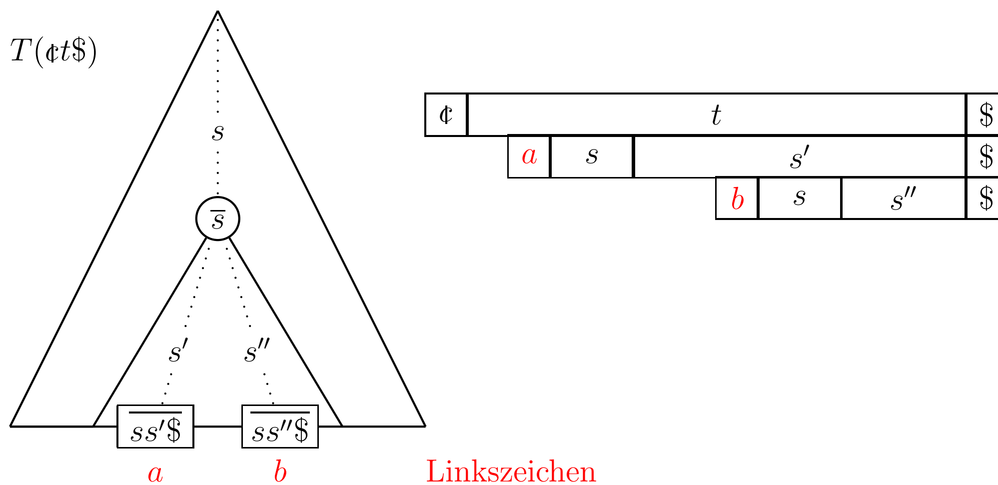


Abbildung 3.4: Skizze: Definition von Linkszeichen

In der Definition ist für einen linksdiversen Knoten v nur gefordert, dass im Teilbaum von v zwei verschiedenen Linkszeichen auftreten müssen. Man kann sich aber leicht überlegen, dass es dann sogar zwei Blätter in den Teilbäumen von zwei verschiedenen Kindern von v mit unterschiedlichem Linkszeichen geben muss.

Theorem 3.10 *Sei $t \in \Sigma^*$, $t' = ¢t\$$ und $T = T(t')$ der Suffix-Baum für t'. Die Zeichenreihe $s \in \Sigma^*$ ist genau dann ein maximaler Repeat von t, wenn der Knoten $\overline{s}$ in T existiert und linksdivers ist.*

Beweis: $\Leftarrow$: Da $\overline{s}$ linksdivers ist, muss es zwei Blätter $\overline{v}$ und $\overline{w}$ im Teilbaum von $\overline{s}$ geben, die unterschiedliche Linkszeichen besitzen.

Wir halten zunächst fest, dass dann $\overline{s}$ zwei verschiedene Kinder besitzen muss, in deren Teilbäumen die Blätter mit den unterschiedlichen Linkszeichen auftreten müssen. Dieser Fall ist in Abbildung 3.5 illustriert.

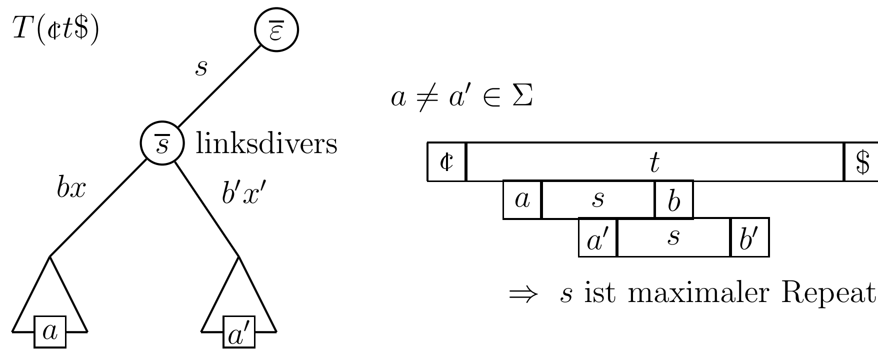


Abbildung 3.5: Skizze: $\overline{s}$ ist linksdivers

Seien bx und $b'x'$ mit $b \neq b' \in \Sigma$ und $x, x' \in \Sigma^*$ die beiden Kantemarkierungen zu den zwei Kindern, die die verschiedenen Linkszeichen im Unterbaum besitzen. Weiter seien $a \neq a' \in \Sigma$ die beiden verschiedenen Linkszeichen, wobei a das Linkszeichen des Blattes ist, der im über die Kante bx erreichbaren Teilbaum liegt. Dann muss sowohl asb als auch $a'sb'$ ein Teilwort von $t' = ¢t\$$ sein. Somit gibt es ein maximales Paar, das genau diese Vorkommen in t' beschreibt und somit ist s ein maximaler Repeat.

$\Rightarrow$: Sei nun s ein maximaler Repeat in t, dann muss es nach Definition ein maximales Paar (i, j, ℓ) geben mit $t_i \cdots t_{i+\ell-1} = t_j \cdots t_{j+\ell-1} = s$, $t'_{i-1} \neq t'_{j-1}$ und $t'_{i+\ell} \neq t'_{j+\ell}$. Somit ist s ein rechtsverzweigendes Teilwort von $¢t\$$ und $\overline{s}$ muss dann ein interner Knoten sein. Da ja $t'_{i-1} \neq t'_{j-1}$ ist, ist das Linkszeichen vom Blatt $\overline{t'_i \cdots t'_{n+1}}$ ungleich dem Linkszeichen von Blatt $\overline{t'_j \cdots t'_{n+1}}$. Da weiter $t'_{i+\ell} \neq t'_{j+\ell}$ ist, tauchen die beiden unterschiedlichen Linkszeichen in zwei verschiedenen Teilbäumen der Kinder von $\overline{s}$ auf und $\overline{s}$ ist somit linksdivers. Dies ist auch in Abbildung 3.6 illustriert. ■

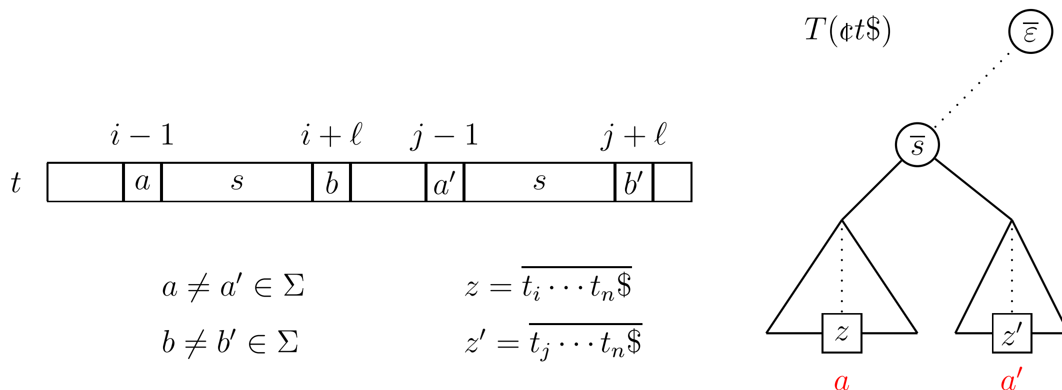


Abbildung 3.6: Skizze: Ein maximales Repeat, das durch sein zugehöriges maximales Paar einen internen Knoten im zugehörigen Suffix-Baum induziert

Damit erhalten auch eine kompakte Darstellung von maximalen Repeats in t. Wir müssen nur den Suffix-Baum für $¢t\$$ konstruieren und dort alle nicht linksdiversen Knoten (inklusive der Blätter) und die dazu inzidenten Kanten entfernen. Man sieht leicht, dass der restlichen Baum zusammenhängend ist und nach dem vorherigen Satz alle maximalen Repeats charakterisiert.

3.1.3 Erkennung maximaler Repeats

Wir müssen nun nur noch einen Algorithmus zur Erkennung linksdiverser Knoten entwickeln.

Wir werden das Problem auch hier wieder mit einer Tiefensuche durch den Suffix-Baum $T = T(¢t\$)$ erledigen. Dazu bestimmen wir an den Blättern das jeweilige Linkszeichen. Hierfür müssen wir für ein Blatt nur wissen, ab welcher Position i das zugehörige Suffix beginnt, dann ist t'_{i-1} das zugehörige Linkszeichen.

An den inneren Knoten bekommen wir von den Kindern entweder die Information zurück, ob das Kind linksdivers ist, oder das Zeichen das an allen Blättern des Teilbaum des Kindes als Linkszeichen notiert ist. Liefern alle Kinder dasselbe Zeichen zurück, so ist der Knoten nicht linksdivers und wir geben dieses Zeichen an seinen Elter zurück. Sind die von den Kindern zurückgelieferten Zeichen unterschiedlich oder ist eines der Kinder linksdivers, so geben wir an den Elter die Information linksdivers zurück.

Die Laufzeit an jedem Blatt ist konstant und an den inneren Knoten proportional zur Anzahl der Kinder des Knotens. Da die Summe der Anzahl der Kinder aller Knoten eines Baumes mit n Knoten gerade $n-1$ ist, ist die Gesamtlaufzeit zur Bestimmung linksdiverser Knoten linear in der Größe des Suffix-Baumes T und somit $O(|t|)$. Wie im vorherigen Abschnitt lassen sich somit die maximalen Repeats sehr leicht ausgeben.

Theorem 3.11 *Sei $t \in \Sigma^n$, dann lassen sich alle maximalen Repeats als Wörter bzw. Referenzen in Zeit $O(n+k)$ ermitteln, wobei k die Anzahl aller Zeichen in den maximalen Repeats bzw. die Anzahl der maximalen Repeats ist.*

Jetzt müssen wir nur alle maximalen Paare (Position und Länge des Repeats in der Zeichenreihe) generieren. Hierfür konstruieren wir für jeden Knoten $\overline{v}$ ein Feld $L_{\overline{v}}[\cdot]$ von Listen, wobei das Feld über die Zeichen des Alphabets Σ indiziert ist und die Listenelemente Positionen innerhalb von t (also Werte aus $[1 : |t|]$) sind. Intuitiv bedeutet dies für einen Knoten $\overline{v}$, dass in seiner Liste für das Zeichen $a \in \Sigma$ die Positionen gespeichert sind, für die a ein Linkszeichen im zugehörigen Teilbaum ist.

An den Blättern erzeugen wir das Feld von Listen wie folgt. Sei b das Linkszeichen des Blattes, dessen zugehöriges Suffix an Position i beginnt. Dann enthalten alle Feldelemente die leere Liste mit Ausnahme für das Zeichen $b \in \Sigma$, die eine einelementige Liste mit Listenelement i enthält. Dies ist in Abbildung 3.7 illustriert.

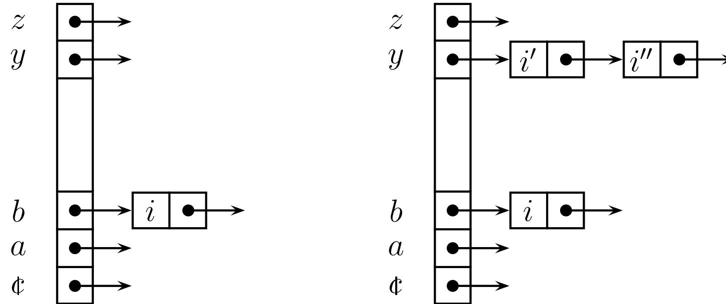


Abbildung 3.7: Skizze: Das Feld der Listen $L_{\overline{v}}$ für ein Blatt $\overline{v}$ mit Linkszeichen $b \in \Sigma$ (rechts) und für einen inneren Knoten (links) mit $\Sigma = \{a, \ldots, z\}$

An den inneren Knoten erzeugen wir dieses Feld, indem wir für jedes Feldelement $a \in \Sigma$ die entsprechenden Listen der Kinder aneinanderhängen. Bevor wir für ein Kind die Listen an die Listen des Knotens $\overline{v}$ anhängen, geben wir erst noch die maximalen Paare wie folgt aus: für jedes Paar von Kindern $\overline{v'} \neq \overline{v''}$ von $\overline{v}$ und für alle $a \neq b \in \Sigma \cup \{¢\}$ erzeuge für alle $i \in L_{\overline{v'}}[a]$ und $j \in L_{\overline{v''}}[b]$ die maximalen Paare $(i, j, |v|)$. Wenn wir das Kind $\overline{v''}$ neu aufnehmen, dann können wir statt der Liste vom Kind $\overline{v'}$ auch die bislang bereits konstruierte Listen von $\overline{v}$ hernehmen.

Für die Laufzeit ist wichtig, dass die Listen konkateniert und nicht kopiert werden, da die Listen ja sehr lang werden können. Prinzipiell muss dazu im Feld jeweils auch noch ein Zeiger auf das letzte Feldelement gespeichert werden. Die Idee der Konkatenation der Felderlisten der Kinder ist (ohne die Zeiger auf das das letzte Listenelement) in Abbildung 3.8 illustriert.

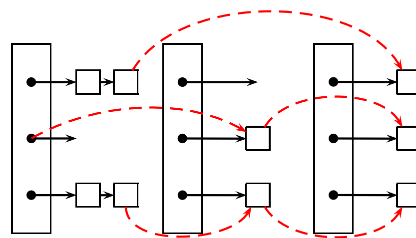


Abbildung 3.8: Skizze: Konkatenation der Felder von Listen zu einem neuen Feld von Listen

Der Aufwand der Konkatenation der Felder von Listen ist wiederum proportional zur Summe der Grade über alle Baumknoten und ist damit wieder linear (allerdings

mit Aufwand $O(|\Sigma|)$ pro Knoten), d.h. insgesamt $O(|\Sigma| \cdot |t|)$. Allerdings kann die Ausgabe der maximalen Paare wiederum mehr Zeit benötigen, jedoch nur konstante Zeit pro maximales Paar. Somit ist die Laufzeit wiederum linear in der Eingabe- und Ausgabegröße, sofern es keine leeren Listen gibt.

Wenn es leere Listen gibt, dann kann konstanter Aufwand entstehen, die nicht durch Ausgabe eine maximalen Paares gedeckt sind. Dies kann den Aufwand um den Faktor $|\Sigma|^2$ erhöhen, da die Anzahl der (nicht künstlichen) Kinder eines Knotens im Suffix-Baum durch $|\Sigma|$ beschränkt ist.

Wenn das Alphabet also groß wird, ist es besser, das Feld als sortierte Liste zu verwalten. Die oben angegebene Aufgaben lassen sich dann wirklich in Zeit $O(|\Sigma| \cdot n + k)$ implementieren. Der Leser möge sich die Details überlegen.

Theorem 3.12 *Sei $t \in \Sigma^n$, dann können alle maximalen Paare in Zeit $O(|\Sigma| \cdot n + k)$ ermittelt werden, wobei k die Anzahl der maximalen Paare ist.*

Auch hier kann man den Algorithmus leicht so modifizieren, dass er nur exakte Paare ausgibt, deren maximalen Repeats eine Mindestlänge aufweisen.

3.1.4 Revers-komplementäre Repeats

In diesem Abschnitt wollen wir uns mit den so genannten revers-komplementären Repeats beschäftigen, die in Genomen auftreten können, wenn das Repeat eigentlich auf dem anderen Strang der DNA-Doppelhelix liegt. Dazu müssen wir erst einmal formal definieren, was wir unter revers-komplementären Repeats verstehen wollen. Wir beginnen mit der Definition von revers-komplementären Sequenzen.

Definition 3.13 *Sei Σ ein Alphabet und sei $\pi \in S(\Sigma)$ eine Permutation auf Σ mit $\pi^2(\sigma) = \sigma$. Für $\sigma \in \Sigma$ ist das über π zugehörige* komplementäre Zeichen $\tilde{\sigma} = \pi(\sigma)$.

Für $w \in \Sigma^*$ ist das über π zugehörige revers-komplementäre Wort $\tilde{w}$ *wie folgt definiert:*

$$\tilde{w} = \begin{cases} \varepsilon & \text{falls } w = \varepsilon, \\ \tilde{v} \cdot \tilde{\sigma} & \text{falls } w = \sigma \cdot v \text{ mit } \sigma \in \Sigma,\ v \in \Sigma^*. \end{cases}$$

Für das biologisch relevante Alphabet $\Sigma = \{A, C, G, T\}$ ist das komplementäre Zeichen wie folgt definiert:

$$\tilde{\sigma} = \pi(\sigma) := \begin{cases} A & \text{falls } \sigma = T, \\ C & \text{falls } \sigma = G, \\ G & \text{falls } \sigma = C, \\ T & \text{falls } \sigma = A. \end{cases}$$

Hierauf basierend können wir nun die so genannten revers-komplementäre Repeats definieren.

Definition 3.14 *Sei $t \in \Sigma^*$. Ein Teilwort s von t heißt* revers-komplementäres Repeat, *wenn es ein Tripel (i, j, ℓ) mit $t_i \cdots t_{i+\ell-1} = s$ und $t_j \cdots t_{j+\ell-1} = \tilde{s}$ gibt.*

Die Lösung zum Auffinden exakter bzw. maximaler revers-komplementärer Repeats ist nun einfach, da wir das Problem auf exakte bzw. maximale Repeats zurückführen können. Sei $t \in \Sigma^*$ die Sequenz in der wir revers-komplementäre Repeats ermitteln wollen. Wir konstruieren zuerst das Wort $t' = ¢t\#\tilde{t}\$$, wobei $¢, \#, \$ \notin \Sigma$ neue Zeichen sind. Ein revers-komplementäres Repeat in t entspricht nun einem normalen Repeat in t'. Dies ist in Abbildung 3.9 schematisch dargestellt.

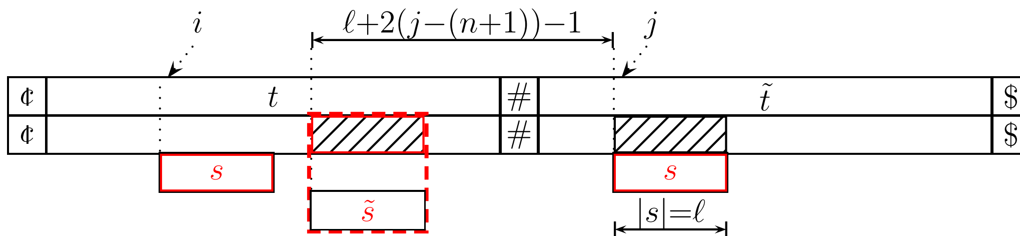


Abbildung 3.9: Skizze: revers-komplementäre Repeats

Wir müssen dabei nur beachten, dass wir auch normale Repeats in t finden (bzw. dann natürlich und nur dann auch in $\tilde{t}$). Wir müssen bei der Ermittlung der Repeats also darauf achten, dass ein Paar (i, j, ℓ) nur dann interessant ist, wenn $i \in [1 : n]$ und $j \in [n + 2 : 2n + 1]$ ist (wobei wir annehmen, dass $t' = t'_0 \cdots t'_{2n+2}$ ist). Als Ausgabe generieren wir dann $(i, j - (\ell + 2(j - (n + 1)) - 1), \ell) = (i, j - \ell - 2j + 2n + 3, \ell)$.

Um nicht mit der Überprüfung von Repeats innerhalb von t (bzw. $\tilde{t}$) aufgehalten zu werden, führen wir in den entsprechenden Algorithmen immer zwei Listen mit. Eine mit den Einträge von Positionen aus $[1 : n]$ und eine mit den Einträgen von Positionen aus $[n + 2 : 2n + 1]$. Bei der Ausgabe müssen wir dann immer nur Paare betrachten, bei denen die Positionen aus den jeweils entgegengesetzten Listen stammen.

Theorem 3.15 *Sei $t \in \Sigma^*$, dann lassen sich alle Paare, die exakte bzw. maximale revers-komplementäre Repeats darstellen, in Zeit $O(|\Sigma| \cdot n + k)$ ermitteln, wobei k die Anzahl der ausgegebenen Paare ist.*

3.2 Tandem-Repeats und Suffix-Bäume

In diesem Abschnitt wollen wir uns mit so genannten Tandem-Repeats beschäftigen, dass sind kurz gesprochen exakte Repeats, die unmittelbar hintereinander in t vorkommen.

3.2.1 Was sind Tandem-Repeats

Zunächst einmal definieren wir formal, was Tandem-Repeats sind.

Definition 3.16 *Für $t \in \Sigma^*$ heißt ein Paar (i, ℓ)* Tandem-Repeat-Paar *in t, wenn $t_i \cdots t_{i+\ell-1} = t_{i+\ell} \cdots t_{i+2\ell-1}$ gilt. Mit $\mathcal{T}(t)$ bezeichnen wir die Menge aller Tandem-Repeat-Paare von t. Das Wort $t_i \cdots t_{i+2\ell-1}$ heißt dann auch* Tandem-Repeat. *Mit $\overline{\mathcal{T}}(t)$ bezeichnen wir die Menge aller Tandem-Repeats von t. Die* Länge eines Tandem-Repeat-Paares (i, ℓ) *bzw.* eines Tandem-Repeats $\alpha\alpha$ *ist 2ℓ bzw. $2|\alpha|$.*

In Abbildung 3.10 ist schematisch ein Tandem-Repeat dargestellt.

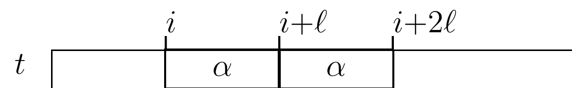


Abbildung 3.10: Skizze: Tandem-Repeat

Wir reden auch manchmal etwas locker von einem Tandem-Repeat anstelle eines Tandem-Repeat-Paares, wenn klar ist, welches zugehörige Tandem-Repeat-Paar hier gemeint ist.

Definition 3.17 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei $t' = t'_0 \cdots t'_{n+1} = ¢t_1 \cdots t_n\$$. Ein Tandem-Repeat-Paar (i, ℓ) von t heißt* rechtsverzweigend *bzw.* linksverzweigend, *wenn $t'_{i+\ell} \neq t'_{i+2\ell}$ bzw. $t'_{i-1} \neq t'_{i+\ell-1}$ gilt.*

Wir sagen auch manchmal etwas locker, dass ein Tandem-Repeat rechtsverzweigend ist, wenn klar ist, welches das zugehörige Tandem-Repeat-Paar ist. Man beachte,

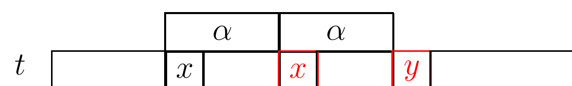


Abbildung 3.11: Skizze: Ein rechtsverzweigendes Tandem-Repeat mit $x \neq y \in \Sigma$

dass es Tandem-Repeats geben kann, für die es sowohl ein Tandem-Repeat-Paar geben kann, das rechtsverzweigend ist, als auch ein anderes, das nicht rechtsverzweigend ist. In Abbildung 3.11 ist schematisch ein rechtsverzweigendes Tandem-Repeat dargestellt.

Definition 3.18 *Sei $t \in \Sigma^*$ und sei (i, ℓ) ein Tandem-Repeat-Paar in t. Ist auch $(i+1, \ell)$ bzw. $(i-1, \ell)$ ein Tandem-Repeat-Paar, so nennen wir das Tandem-Repeat-Paar (i, ℓ) eine* Linksrotation *von $(i+1, \ell)$ bzw.* Rechtsrotation *von $(i-1, \ell)$.*

Mit Hilfe dieser Definition können wir uns beim Auffinden von Tandem-Repeat-Paaren auf rechtsverzweigende (oder auch linksverzweigende) beschränken, wie die folgende Beobachtung zeigt.

Beobachtung 3.19 *Jedes nicht rechts- bzw. linksverzweigende Tandem-Repeat-Paar (i, ℓ) ist eine Linksrotation des Tandem-Repeat-Paares $(i+1, \ell)$ bzw. eine Rechtsrotation des Tandem-Repeat-Paares $(i-1, \ell)$.*

Diese Beobachtung ist in der folgenden Abbildung 3.12 noch einmal illustriert.

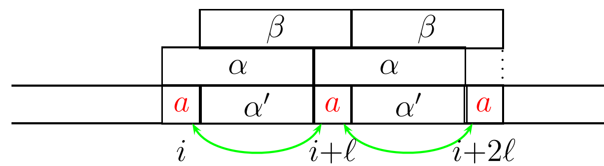


Abbildung 3.12: Skizze: Nicht rechtsverzweigendes Tandem-Repeat $\alpha\alpha$ als Linksrotation eines anderen Tandem-Repeats $\beta\beta$

Der Vollständigkeit halber definieren wir bereits an dieser Stelle noch so genannte primitive Tandem-Repeats.

Definition 3.20 *Eine Zeichenreihe $s \in \Sigma^+$ heißt* primitiv*, wenn aus $s = u^k$ für ein $u \in \Sigma^+$ und ein $k \in \mathbb{N}$ folgt, dass $k = 1$ ist.*

Ein Tandem-Repeat $\alpha\alpha$ heißt primitiv*, wenn α primitiv ist.*

Der Sinn hinter dieser Definition ist es, eine weitere Einschränkung auf interessante Tandem-Repeats zu besitzen. Nichtprimitive Tandem-Repeats bestehen ihrerseits aus einer Konkatenation gleicher Teilwörter und enthalten ihrerseits kürzere primitive Tandem-Repeats, die Anfänge von mehrfachen Wiederholungen sind, wie z.B. α^4. Solche mehrfach gekoppelten Repeats werden in der Literatur oft auch *Tandem-Arrays* genannt.

3.2.2 Eigenschaften von Tandem-Repeats

In diesem Abschnitt wollen wir einige fundamentale Beziehungen von (rechtsverzweigenden) Tandem-Repeats in t und dem zu t gehörigen Suffix-Baum aufstellen. Zunächst einmal wiederholen wir die Definition des im Folgenden wichtigen Begriffs der Worttiefe eines Knotens im Suffix-Baum.

Definition 3.21 *Sei $t \in \Sigma^*$ und sei $T = T(t\$)$ der Suffix-Baum für $t\$$. Für einen Knoten $v \in V(T)$ definieren wir seine* Worttiefe *als $|path(v)|$.*

Für das Folgende ist auch der Begriff der Blattlisten (engl. leaf lists) von fundamentaler Bedeutung, die wir beim Algorithmus zur Erkennung exakter Repeats schon kennen gelernt und dort auch eingesetzt haben, aber bislang noch nicht formal definiert haben.

Definition 3.22 *Sei $t \in \Sigma^*$ und sei $T = T(t\$)$ der Suffix-Baum zu $t\$$. Für ein Blatt $\overline{v} \in V(T)$ ist die* Blattliste *$LL(\overline{v})$ als die einelementige Menge $\{|t\$| - |v| + 1\}$ definiert (d.h. die Indexposition, an der der zugehörige Suffix in t auftritt). Die* Blattliste *$LL(\overline{v})$ eines inneren Knotens $\overline{v} \in V(T)$ ist definiert durch*

$$LL(\overline{v}) := \bigcup_{\substack{\overline{w} \in V(T) \\ (\overline{v},\overline{w}) \in E(T)}} LL(\overline{w})$$

(d.h. die Menge aller Indexposition von Suffixen, deren zugehörigen Blätter sich im Teilbaum von $\overline{v}$ befinden).

Mit Hilfe dieser Begriffe können wir nun eine Charakterisierung von Tandem-Repeats durch Knoten im zugehörigen Suffix-Baum angeben.

Lemma 3.23 *Sei $t \in \Sigma^n$ und sei $i < j \in [1 : n]$ sowie $\ell := j - i$. Dann sind folgende Aussagen äquivalent:*

1. *(i, ℓ) ist ein Tandem-Repeat-Paar in t.*
2. *Es existiert ein Konten $\overline{v} \in V(T(t\$))$ mit $|v| \geq \ell$ und $i, j \in LL(\overline{v})$.*

Beweis: 1 ⇒ 2 : Nach Voraussetzung ist also (i, ℓ) ein Tandem-Repeat-Paar in t. Dies bedeutet, dass $t_i \cdots t_{i+\ell-1} = t_{i+\ell} \cdots t_{i+2\ell-1}$ gilt. Dies ist in Abbildung 3.13 illustriert.

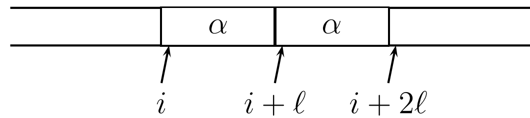


Abbildung 3.13: Skizze: (i, ℓ) ist ein Tandem-Repeat-Paar in t

Sei also $\alpha\alpha$ das zu (i, ℓ) gehörige Tandem-Repeat. Somit beginnt das Suffix an Position i sowie das Suffix an Position $j := i + \ell$ jeweils mit α. Sei weiter β so gewählt, dass $\overline{\alpha\beta}$ der Knoten in $T(t\$)$ ist, an dem sich die Pfade von der Wurzel zu den Blättern von $t_i \cdots t_n\$$ und $t_{i+\ell} \cdots t_n\$$ trennen, siehe hierzu auch Abbildung 3.14.

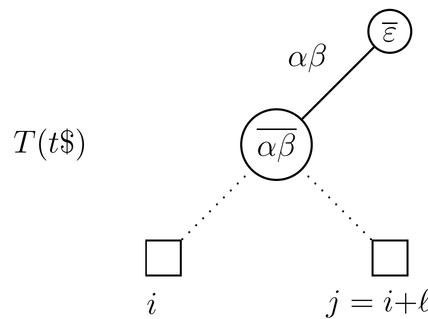


Abbildung 3.14: Skizze: Tandem-Repeat $\alpha\alpha$ und der Suffix-Baum $T(t\$)$

Dann befinden sich jedoch i und $j = i+\ell$ in der Blattliste $LL(\overline{\alpha\beta})$ und die Worttiefe von $\overline{\alpha\beta}$ ist gleich $|\alpha\beta| \geq |\alpha| = \ell$.

2 ⇒ 1 : Sei also v ein Knoten von $T(t\$)$ mit Worttiefe mindestens ℓ so gewählt, dass $i, j \in LL(v)$ gilt. Weiterhin sei $\text{path}(v) = \alpha\beta$ mit $|\alpha| = \ell$ und $\beta \in \Sigma^*$. Dieser Fall ist in Abbildung 3.15 illustriert.

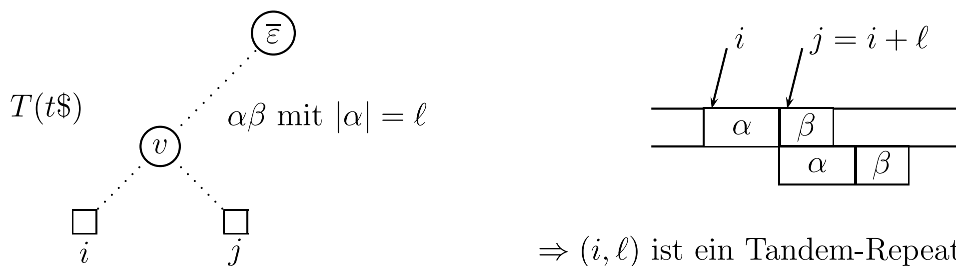


Abbildung 3.15: Skizze: Knoten v mit Worttiefe mindestens ℓ und $i, j \in LL(v)$

Daraus folgt nun sofort, dass $t_i \cdots t_{i+|\alpha\beta|-1} = \alpha\beta$ und $t_j \cdots t_{j+|\alpha\beta|-1} = \alpha\beta$. Da $|\alpha| = \ell$ und $\ell = j - i$ ist, folgt, dass sich die beiden Vorkommen in t überlappen müssen und $\alpha\alpha$ somit ein Tandem-Repeat zum Tandem-Repeat-Paar (i, ℓ) sein muss. Dies ist im

linken Teil der Abbildung 3.15 besonders gut zu erkennen. Damit ist die Behauptung bewiesen. ■

Wir können auch eine analoge Charakterisierung für rechtsverzweigende Tandem-Repeats angeben.

Lemma 3.24 *Sei $t \in \Sigma^n$ und sei $i < j \in [1 : n]$ sowie $\ell := j - i > 0$. Dann sind folgende Aussagen äquivalent:*

1. *Das Paar (i, ℓ) ist ein rechtsverzweigendes Tandem-Repeat-Paar.*
2. *Es existiert ein Knoten $\overline{v} \in V(T(t\$))$ mit $|v| = \ell$ und $i, j \in LL(\overline{v})$. Weiterhin gilt für alle Knoten $\overline{w} \in V(T(t\$))$ mit $|w| > \ell$, dass nicht sowohl $i \in LL(\overline{w})$ als auch $j \in LL(\overline{w})$ gilt.*

Beweis: Der Beweis ist analog zum Beweis in Lemma 3.23 und bleibt dem Leser zur Übung überlassen. ■

3.2.3 Algorithmus von Stoye und Gusfield

Aus diesem Lemma 3.24 lässt sich sofort der folgende, in Abbildung 3.16 angegebene Algorithmus von Stoye und Gusfield für das Auffinden rechtsverzweigender Tandem-Repeat-Paare herleiten. Dabei werden die rechtsverzweigenden Tandem-Repeat-Paare explizit durch die Abfrage $t_{i+|\mathrm{path}(v)|} \neq t_{i+2|\mathrm{path}(v)|}$ herausgefiltert. Aufgrund der Tatsache, dass die Anfangsposition eines rechtsverzweigendes Tandem-Repeat-Paares nur an einem Knoten aufgefunden werden kann (siehe Lemma 3.24), wird jedes rechtsverzweigendes Tandem-Repeat-Paar genau einmal ausgegeben.

03.12.20

```
TandemRepeats (string t)
begin
    tree T := SuffixTree(t$);
    foreach (v ∈ V(T)) do                                   /* using DFS */
        // after returning from all children
        // creating leaf lists
        LL(v) := ∅;
        foreach ((v, w) ∈ E(T)) do
            LL(v) := LL(v) ∪ LL(w);
        if (LL(v) = ∅) then
            LL(v) := {|t| + 2 − |path(v)|};
        ℓ := |path(v)|;
        foreach (i ∈ LL(v)) do
            if ((i + ℓ ∈ LL(v)) && (t_{i+ℓ} ≠ t_{i+2ℓ})) then
                output (i, ℓ);
end
```

Abbildung 3.16: Algorithmus von Stoye und Gusfield

A Literaturhinweise

A.1 Lehrbücher zur Vorlesung

D. Adjeroh, T. Bell, A. Mukherjee: *The Burrows-Wheeler Transform*, Springer, 2008

S. Aluru (Ed.): *Handbook of Computational Molecular Biology*; Chapman and Hall/CRC, 2006.

H.-J. Böckenhauer, D. Bongartz: *Algorithmische Grundlagen der Bioinformatik: Modelle, Methoden und Komplexität*; Teubner, 2003.

G. Fertin, A. Labarre, I. Rusu, E. Tannier, S. Vialette: *Combinatorics of Genome Rearrangements*; MIT Press, 2009.

D. Gusfield: *Algorithms on Strings, Trees, and Sequences — Computer Science and Computational Biology*; Cambridge University Press, 1997.

M. Lothaire: *Applied Combinatorics on Words*, Encyclopedia of Mathematics and Its Applications, Cambridge University Press, 2005.

V. Mäkinen, D. Belazzougui, F. Cunial, A.I. Tomescu: *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*, Cambridge University Press, 2015.

E. Ohlebusch: *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*, Oldenbusch Verlag, 2013.

P.A. Pevzner: *Computational Molecular Biology — An Algorithmic Approach*; MIT Press, 2000.

J.C. Setubal, J. Meidanis: *Introduction to Computational Molecular Biology*; PWS Publishing Company, 1997.

A.2 Skripten anderer Universitäten

J. Fischer: *Text-Indexierung und Information Retrieval*, Technische Universität Dortmund, 2015.
`ls11-www.cs.tu-dortmund.de/_media/fischer/teaching/tir-ws2014/script-tir-ws14.pdf`

S. Kurtz: *Lecture Notes for Foundations of Sequence Analysis*, Universität Bielefeld, 2001.
citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.7142&rep=rep1&type=pdf

A.3 Originalarbeiten

A.3.1 Optimal Scoring Subsequences

K.-M. Chung, H.-I. Lu: An Optimal Algorithm for the Maximum-Density Segment Problem, *SIAM Journal on Computing*, Vol. 34, No. 2, 373–387, 2004.
DOI: 10.1137/S0097539704440430

M. Csűrös: Maximum-Scoring Segment Sets, *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, Vol. 1, No. 4, 139–150, 2004.
DOI: 10.1109/TCBB.2004.43

P. Fariselli, M. Finelli, D. Marchignoli, P.L. Martelli, I. Rossi, R. Casadio: MaxSubSeq: An Algorithm for Segment-Length Optimization. The Case Study of the Transmembrane Spanning Segments, *Bioinformatics*, Vol. 19, 500–505, 2003.
DOI: 10.1093/bioinformatics/btg023

M.H. Goldwasser, M.-Y. Kao, H.-I. Lu: Linear-Time Algorithms for Computing Maximum-Density Sequence Segments with Bioinformatics Applications, *Journal of Computer and System Sciences*, Vol.70, No. 2, 128–144, 2005.
DOI: 10.1016/j.jcss.2004.08.001

S.K. Kim: Linear-Time Algorithm for Finding a Maximum-Density Segment of a Sequence, *Information Processing Leter*, Vol. 86, 339–342, 2003.
DOI: 10.1016/S0020-0190(03)00225-4

Y.-L. Lin, T. Jiang, K.-M. Chao: Efficient Algorithms for Locating the Length-Constrained Heaviest Segments with Applications to Biomolecular Sequence Analysis, *Journal of Computer and System Sciences*, Vol. 65, 570–586, 2002.
DOI: 10.1016/S0022-0000(02)00010-7

W.L. Ruzzo, M. Tompa: A Linear Time Algorithm for Finding All Maximal Scoring Subsequences, *Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology (ISMB'99)*, 234–241, 1999.

A.3.2 Suffix-Trees

R. Giegerich, S. Kurtz, J. Stoye: Efficient Implementation of Lazy Suffix Trees, *Software — Practice and Experience*, Vol. 33, 1035–1049, 2003.
DOI: 10.1002/spe.535

M. Maaß: *Suffix Trees and Their Applications*, Ausarbeitung von der Ferienakademie, Kurs 2, Bäume: Algorithmik und Kombinatorik, 1999.
www14.in.tum.de/konferenzen/Ferienakademie99/

E.M. McCreight: A Space-Economical Suffix Tree Construction Algorithm; *Journal of the ACM*, Vol. 23, 262–272, 1976.
DOI: 10.1145/321941.321946

E. Ukkonen: On-Line Construction of Suffix Trees, *Algorithmica*, Vol. 14, 149–260, 1995.
DOI: 10.1007/BF01206331

A.3.3 Repeats

A.S. Fraenkel, J. Simpson: How Many Squares Can a String Contain?, *Journal of Combinatorial Theory, Series A*, Vol. 82, 112–120, 1998.
DOI: 10.1006/jcta.1997.2843

D. Gusfield, J. Stoye: Linear Time Algorithm for Finding and Representing All the Tandem Repeats in a String, *Journal of Computer and System Sciences*, Vol. 69, 525–546, 2004; see also *Technical Report* CSE-98-4, Computer Science Department, UC Davis, 1998.
DOI: 10.1016/j.jcss.2004.03.004

R. Kolpakov, G. Kucherov: Finding Maximal Repetitions in a Word in Linear Time, *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS'99)*, 596–604, 1999.
DOI: 10.1109/SFFCS.1999.814634

R. Kolpakov, G. Kucherov: On Maximal Repetitions in Words, *Proceedings of the 12th International Symposium on Fundamentals of Computation Theory (FCT'99)*, Lecture Notes in Computer Science, Vol. 1684, 374–385, 1999.
DOI: 10.1007/3-540-48321-7_31

R. Kolpakov, G. Kucherov: Finding Approximate Repetitions Under Hamming Distance, *Theoretical Computer Science*, Vol. 303, 135–156, 2003.
DOI: 10.1016/S0304-3975(02)00448-6

G.M. Landau, J.P. Schmidt: An Algorithm for Approximate Tandem Repeats, *Proceedings of the 4th Symposium on Combinatorial Pattern Matching (CPM'93)*, Lecture Notes in Computer Science, Vol. 684, 120–133, 1993. DOI: 10.1007/BFb0029801

G.M. Landau, J.P. Schmidt, D. Sokol: An Algorithm for Approximate Tandem Repeats, *Journal of Computational Biology*, Vol. 8, No. 1, 1–18, 2001. DOI: 10.1089/106652701300099038

M.G. Main, R.J. Lorentz: An $O(n \log n)$ Algorithm for Finding All Repetitions in a String, *Journal of Algorithms*, Vol. 5, No. 3, 422–432, 1984. DOI: 10.1016/0196-6774(84)90021-X

J. Stoye, D. Gusfield: Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree, *Theoretical Computer Science*, Vol. 270, 843–856, January 2002. DOI: 10.1016/S0304-3975(01)00121-9

A.3.4 Lowest Common Ancestors and Range Minimum Queries

S. Alstrup, C. Gavoille, H. Kaplan, T. Rauhe: Nearest Common Ancestors: A Survey and a New Distributed Algorithm, *Theory of Computing Systems*, Vol. 37, No. 3, 441–456, 2004. DOI: 10.1007/s00224-004-1155-5

M.A. Bender, M. Farach-Colton: The LCA Problem Revisited, *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN'00)*, Lecture Notes in Computer Science, Vol. 1776, 88–94, 2000. DOI: 10.1007/10719839_9

O. Berkman, U. Vishkin: Recursive Star-Tree Parallel Data Structure, *SIAM Journal on Computing*, Vol. 22, 221-242, 1993. DOI: 10.1137/0222017

J. Fischer, V. Heun. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays, *SIAM Journal on Computing*, Vol. 40, No. 2, 465–492, 2011. DOI: 10.1137/090779759

B. Schieber, U. Vishkin: On Finding Lowest Common Ancestors: Simplification and Parallelization, *SIAM Journal on Computing*, Vol. 17, 1253–1262, 1988. DOI: 10.1137/0217079

A.3.5 Construction of Suffix-Arrays

S. Burkhardt, J. Kärkkäinen: Fast Lightweight Suffix Array Construction and Checking, *Proceedings of the 14th Symposium on Combinatorial Pattern Matching (CPM'03)*, Lecture Notes in Computer Science, Vol. 2676, 55–69, 2003.
DOI: 10.1007/3-540-44888-8_5

S. Gog, T. Beller, A. Moffat. M. Petri: From Theory to Practice: Plug and Play with Succinct Data Structures, *Proceedinsg of the 13th International Symposium on Experimental Algorithms (SEA 2014)*, Lecture Notes in Computer Science, Vol. 8504, 326–337, 2014.
DOI: 10.1007/978-3-319-07959-2_28
SDSL Homepage: algo2.iti.kit.edu/gog/docs/html/index.html

D.K. Kim, J.S. Sim, H. Park, K. Park: Linear-Time Construction of Suffix Arrays, *Proceedings of the 14th Symposium on Combinatorial Pattern Matching (CPM'03)*, Lecture Notes in Computer Science, Vol. 2676, 186–199, 2003.
DOI: 10.1007/3-540-44888-8_14

P. Ko, A. Aluru: Space Efficient Linear Time Construction of Suffix Arrays, *Proceedings of the 14th Symposium on Combinatorial Pattern Matching (CPM'03)*, Lecture Notes in Computer Science, Vol. 2676, 200–210, 2003.
DOI: 10.1007/3-540-44888-8_15

J. Kärkkäinen, P. Sanders: Simple Linear Work Suffix Array Construction, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, Lecture Notes in Computer Science, Vol. 2719, 943–955, 2003.
DOI: 10.1007/3-540-45061-0_73

G. Nong, S. Zhang, W.H. Chan: Two Efficient Algorithms for Linear Time Suffix Array Construction, *IEEE Transaction on Computers*, Vol 60, No. 10, 1471–1484, 2011.
DOI: 10.1109/TC.2010.188

U. Manber, G. Myers: Suffix Arrays: A New Method for On-Line String Searches, *SIAM Journal on Computing*, Vol. 22, 935–948, 1993.
DOI: 10.1137/0222058

Y. Mori: Implementation of SAIS in Different Programming Languages (C/C++, C#, Java). Web: sites.google.com/site/yuta256/sais vom 15.12.2017, zuletzt zugegriffen am 08.01.2018.

A.3.6 Applications of Suffix-Arrays

M.I. Abouelhoda, S. Kurtz, E. Ohlebusch: The Enhanced Suffix Array and Its Applications to Genome Analysis, *Proceedings of the 2nd Workshop on Algorithms in Bioinformatics (WABI'02)*, Lecture Notes in Computer Science, Vol. 2452, 449–463, 2002.
DOI: 10.1007/3-540-45784-4_35

M.I. Abouelhoda, E. Ohlebusch, S. Kurtz: Optimal Exact String Matching Based on Suffix Arrays, *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE'02)*, Lecture Notes in Computer Science, Vol. 2476, 31–43, 2002.
DOI: 10.1007/3-540-45735-6_4

M.I.Abouelhoda, S. Kurtz, E. Ohlebusch: Replacing Suffix Trees with Enhanced Suffix Arrays, *Journal of Discrete Algorithms*, Vol. 2, 53–86, 2004.
DOI: 10.1016/S1570-8667(03)00065-0

M. Burrows, D.J. Wheeler: A Block-Sorting Lossless data Compression Algorithm; *Research Report*, Digital Research Center, SRC-Report 124, 1994.
www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html

P. Ferragina, G. Manzini: Indexing Compressed Text; *Journal of the ACM*, Vol. 52, Issue 4, 552–581, 2005
DOI: 10.1145/1082036.1082039

J. Fischer: Combined Data Structure for Previous- and Next-Smaller-Values; *Theoretical Computer Science*, Vol. 412, Issue 22, 2451–2456, 2011.
DOI: 10.1016/j.tcs.2011.01.036

J. Fischer, V. Heun: A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array, *Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE'07)*, Lecture Notes in Computer Science, Vol. 4614, 459–470, Springer, 2007.
DOI: 10.1007/978-3-540-74450-4_41

T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications, *Proceedings of the 12th Symposium on Combinatorial Pattern Matching (CPM'01)*, Lecture Notes in Computer Science, Vol. 2089, 181–192, 2001.
DOI: 10.1007/3-540-48194-X_17

G. Manzini: Two Space Saving Tricks for Linear Time LCP Array Computation, *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT'04)*, Lecture Notes in Computer Science, Vol. 3111, 372–383, 2004.
DOI: 10.1007/b98413

G. Navarro, V. Mäkinen: Compressed Full-Text Indexes, *ACM Computing Surveys*, Vol. 39, No. 1, 2007.
DOI: 10.1145/1216370.1216372

K. Sadakane: Succinct Representations of LCP Information and Improvements in the Compressed Suffix Arrays, *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, 225-232, 2002.

K. Sadakane: Compressed Suffix Trees with Full Functionality, *Theory of Computing Systems*, Vol. 41, No. 4, 589–607, 2007.
DOI: 10.1007/s00224-006-1198-x

A.3.7 Sorting by Reversals

V. Bafna, P.A. Pevzner: Genome Rearrangements and Sorting by Reversals, *SIAM Journal on Computing*, Vol. 25, 272–289, 1996.
DOI: 10.1137/S0097539793250627

P. Berman, S. Hannenhalli, M. Karpinski: A 1.375-Approximation Algorithm for Sorting by Reversals, *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02)*, Lecture Notes in Computer Science, Vol. 2461, 200–210, 2002.
DOI: 10.1007/3-540-45749-6_21

P. Berman, M. Karpinski: On Some Tighter Inapproximability Results, *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP'99)*, Lecture Notes in Computer Science, Vol. 1644, 200–209, 1999.
DOI: 10.1007/3-540-48523-6_17

D. Christie: A 3/2-Approximation Algorithms for Sorting by Reversals, *Proceedings of the 9th ACM Symposium on Discrete Algorithms (SODA'98)*, 244–252, 1998.

A.3.8 Sorting by Oriented Reversals

D.A. Bader, N.M.E. Moret, M. Yan: A Linear-Time Algorithm for Computing Inversion Distance Between Signed Permutations With an Experimental Study, *Journal of Computational Biology*, Vol. 8, No. 5, 483–491, 2001.
DOI: 10.1089/106652701753216503

A. Bergeron: A Very Elementary Presentation of the Hannenhalli-Pevzner Theory, *Discrete Applied Mathematics*, Vol. 146, No. 2, 134–145, 2005.
DOI: 10.1016/j.dam.2004.04.010

A. Bergeron, J. Mixtacki, J. Stoye: Reversal Distance without Hurdles and Fortresses, *Proceedings of the 15th Symposium on Combinatorial Pattern Matching (CPM'04)*, Lecture Notes in Computer Science, Vol. 3109, 388–399, 2004.
DOI: 10.1007/b98377

P. Berman, S. Hannehalli: Faster Sorting by Reversals, *Proceedings of the 7th Symposium on Combinatorial Pattern Matching (CPM'96)*, Lecture Notes in Computer Science, Vol. 1075, 168–185, 1996.
DOI: 10.1007/3-540-61258-0_14

S. Hannehalli, P. Pevzner: Transforming Cabbage into Turnip: Polynomial Algorithm for Sorting Signed Permutations by Reversals, *Journal of the ACM*, Vol. 46, 1–27, 1999; also in *Proceedings of the 27th Annual ACM Symposium on Computing (STOC'95)*, 178–189, 1995.
DOI: 10.1145/300515.300516

S. Hannehalli, P. Pevzner: To Cut ... or Not to Cut (Applications of Comparative Physical Maps in Molecular Evolution), *Proceedings of the 7th ACM Symposium on Discrete Algorithms (SODA'96)*, 304–313, 1996.

H. Kaplan, R. Shamir, R. Tarjan: Faster and Simpler Algorithm for Sorting Signed Permutations by Reversals, *SIAM Journal on Computing*, Vol. 29, 880–892, 1999; also in *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, 344–351, 1997.
DOI: 10.1137/S0097539798334207

A.C. Siepel: An Algorithm to Enumerate all Sorting Reversals, *Journal Computational Biology*, Vol. 10, 575–597, 2003; also in *Proceedings of the 6th Annual International Conference on Computational Biology (RECOMB'02)*, 281–290, 2002.
DOI: 10.1145/565196.565233

A.3.9 Sorting by Transpositions

V. Bafna, P.A. Pevzner: Sorting by Transpositions, *SIAM Journal on Discrete Mathematics*, Vol. 11, No. 2, 224–240, 1998.
DOI: 10.1137/S089548019528280X

L.F.I. Cunha, L.A.B. Kowada, R. de A. Hausen, C.M.H. de Figueiredo: A faster 1.375-approximation algorithm for sorting by transpositions, *Journal on Computational Biology*, Vol. 22, No. 11, 1044-56, 2015.
DOI: 10.1089/cmb.2014.0298

I. Elias, T. Hartman: A 1.375-Approximation Algorithm for Sorting by Transpositions, *Proceedings of the Fifth Workshop on Algorithms in Bioinformatics (WABI'05)*, Lecture Notes in Computer Science, Vol. 3692, 204–215, 2005. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, Vol. 3,No. 4, 369–379, 2006.
DOI: 10.1109/TCBB.2006.44

T. Hartman: A Simpler 1.5-Approximation Algorithms for Sorting by Transpositions, *Proceedings of the 14th Symposium on Combinatorial Pattern Matching (CPM'03)*, Lecture Notes in Computer Science, Vol. 2676, 156–169, 2003.
DOI: 10.1007/3-540-44888-8_12

A.3.10 Sorting by Transversals

M. Bader, E. Ohlebusch: Sorting by Weighted Reversals, Transpositions, and Inverted Transpositions, *Proceedings of the 10th Annual International Conference on Research in Computational Molecular Biology (RECOMB'06)*, Lecture Notes in Computer Science, Vol. 3909, 563–577, Springer, 2006.
DOI: 10.1007/11732990_46

N. Eriksen: $(1 + \varepsilon)$-Approximation of Sorting by Reversals and Transpositions, *Theoretical Computer Science*, Vol. 289, 517–529, 2002.
DOI: 10.1016/S0304-3975(01)00338-3

Q.-P. Gu, S. Peng, I.H. Sudborough: A 2-Approximation Algorithm for Genome Rearrangements by Reversals and Transpositions, *Theoretical Computer Science*, Vol. 210, 327–339, 1999.
DOI: 10.1016/S0304-3975(98)00092-9

T. Hartman, R. Sharan: A 1.5-Approximation Algorithm for Sorting by Transpositions and Transreversals, *Proceedings of the 4th Workshop on Algorithms in Bioinformatics (WABI'04)*, Lecture Notes in Computer Science, Vol. 3240, 50–61, Springer, 2004.
DOI: 10.1007/978-3-540-30219-3_5

T. Hartman, R.Sharan: A 1.5-Approximation Algorithm for Sorting by Transpositions and Transreversals. *Journal of Computer and System Sciences*, Vol. 70, No. 3, 300–320, 2005.
DOI: 10.1016/j.jcss.2004.12.006

G-H. Lin, G. Xue: Signed Genome Rearrangements by Reversals and Transpositions: Models and Approximations, *Theoretical Computer Science*, Vol. 259, 513–531, 2001
DOI: 10.1016/S0304-3975(00)00038-4

A. Rahmana, S. Shatabdaa, M. Hasan: An approximation algorithm for sorting by reversals and transpositions; *Journal of Discrete Algorithms*, Vol. 6, No. 3, 449–45, 2008.
DOI: 10.1016/j.jda.2007.09.002

A.3.11 Erweiterungen zu Genome Rearrangements

M.A. Bender, D. Ge, S. He, H. Hu, R.Y. Pinter, S. Skiena, F. Swidan: Improved Bounds on Sorting with Length-Weighted Reversals, *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'04*, 919–928, 2004.

A. Bergeron, J. Mixtacki, J. Stoye: On Sorting by Translocations, *Proceedings of the 9th Annual International Conference on Computational Biology (RECOMB'05)*, Lecture Notes in Computer Science, Vol. 3500, 615–629, Springer, 2005.
DOI: 10.1007/11415770_47

A. Bergeron, J. Mixtacki, J. Stoye: A Unifying View of Genome Rearrangements, *Proceedings of the 6th International Workshop on Algorithms in Bioinformatics, (WABI'06)*, Lecture Notes in Computer Science, Vol. 4175, 163–173, Springer, 2006.
DOI: 10.1007/11851561_16

D. Bryant: A Lower Bound for the Breakpoint Phylogeny Problem, *Proceedings of the 11th Workshop on Combinatorial Pattern Matching, CPM'00*, 235–247, 2000.
DOI: 10.1007/3-540-45123-4_21

A. Caprara: Formulations and Hardness of Multiple Sorting by Reversals, *Proceedings of the 3rd Annual International Conference on Computational Biology (RECOMB'99)*, 84–93, 1999.
DOI: 10.1145/299432.299461

T. Chen, S.S. Skiena: Sorting with Fixed-Length Reversals, *Discrete Applied Mathematics*, Vol .71, 269–295, 1996.
DOI: 10.1016/S0166-218X(96)00069-8

D.A. Christie, R.W. Irving: Sorting Strings by Reversals and by Transpositions, *SIAM Journal on Discrete Mathematics*, Vol. 14, 193–206, 2001.
DOI: 10.1137/S0895480197331995

D.A. Christie: Sorting Permutations by Block-Interchanges, *Information Processing Letters*, Vol. 60, 165–169, 1996.
DOI: 10.1016/S0020-0190(96)00155-X

D.S. Cohen, M. Blum: On the Problem of Sorting Burnt Pancakes, *Discrete Applied Mathematics*, Vol. 61, No. 2, 105–120, 1995.
DOI: 10.1016/0166-218X(94)00009-3

B. DasGupta, T. Jiang, S. Kannan, M. Li, E. Sweedyk: On the Complexity and Approximation of Syntenic Distance, *Discrete Applied Mathematics*, Vol. 88, 59–82, 1998.
DOI: 10.1016/S0166-218X(98)00066-3

N. El-Mabrouk, D. Sankoff: The Reconstruction of Doubled Genomes, *SIAM Journal on Computing*, Vol. 32, 754–792, 2003.
DOI: 10.1137/S0097539700377177

W. Gates, C. Papadimitriou: Bounds for Sorting by Prefix Reversal, *Discrete Mathematics*, Vol. 27, 47–57, 1979.
DOI: 10.1016/0012-365X(79)90068-2

S. Hannehalli: Polynomial-Time Algorithm for Computing Translocation Distance Between Genomes, *Discrete Applied Mathematics*, Vol .71, 137–151, 1996.
DOI: 10.1016/S0166-218X(96)00061-3

S. Hannenhalli, P.A. Pevzner: To cut ... or not to cut (applications of comparative physical maps in molecular evolution) , *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 96)*, 304–313, 1996.

S. Hannenhalli, P.A. Pevzner: Transforming Men into Mice (Polynomial Algorithm for Genomic Distance), *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science (FOCS'95)*, 581–592, 1995.
DOI: 10.1145/640075.640108

L.S. Heath, J.P. Vergara: Sorting by Bounded Block-Moves, *Discrete Applied Mathematics*, Vol. 88, 181–206, 1998.
DOI: 10.1016/S0166-218X(98)00072-9

J. Kleinberg, D. Liben-Nowell: The Syntenic Diameter of the Space of n-Chromosome Genomes, in *Comparative Genomics*, David Sankoff and Joseph H. Nadeau (Eds.), Kluwer Academic Press, 2000.

J. Meidanis, Z. Dias: Genome Rearrangements Distance by Fusion, Fission, and Transposition is Easy, *Proceedings of the 8th Symposium on String Processing and Information Retrieval, SPIRE'01*, 250–253, 2001
DOI: 10.1109/SPIRE.2001.989776

R.Y. Pinter, S. Skiena: Genomic Sorting with Length-Weighted Reversals, *Genome Informatics*, Vol. 13, 103–111, 2002.

Index B

A
a-MSS, 17
aktives Suffix, 74
Alignment-Distanz, 85
All Maximal Scoring Subsequences, 10, 11
AMSS, 10, 11
atomic suffix tree, 61

B
BAMSS, 27
Blattliste, 98
BMSS, 29
Bounded All Maximum Scoring Subsequences, 27
Bounded Maximal Scoring Subsequence, 29

C
compact suffix tree, 61

D
Darstellung eines Wortes, 58
Divide-and-Conquer, 5
dynamische Programmierung, 4

E
EDIT-Distanz, 85
exaktes Paar, 85
exaktes Repeat, 85

F
fallend rechtsschiefe Partition, 34
Folge
- linksnegative, 30
- linksschiefe, 51
- rechtsschiefe, 34

G
GC-reiche Regionen, 3
Geburtstagsparadoxon, 65

H
Hamming-Distanz, 85
Hashfunktion, 65
Hashing, 65

I
iterierter rechtsschiefer Zeiger, 43

K
k-difference Repeat, 85
k-mismatch Repeat, 85
kanonische Lokation, 79
Kollision, 65
kompakter Σ^+-Baum, 58
kompaktifizierter Trie, 58
komplementäres Zeichen, 94
konservierte Regionen, 3

L
Länge eines Tandem-Repeat-Paares, 96
Länge eines Tandem-Repeats, 96
linksdivers, 90
linksnegative Folge, 30
linksnegativer Zeiger, 31
Linksrotation, 97
linksschiefe Folge, 51
linksverzweigend, 96
linksverzweigendes Tandem-Repeat-Paar, 96
Linkszeichen
- einer Position, 90
- eines Blattes, 90

Lokation, 78
- kanonische, 79
- offene, 79

M

MASS, 34
Maximal Average Scoring Subsequence, 34, 45
maximal bewertete Teilfolge, 10, 11
Maximal Scoring Subsequence, 1
maximaler Repeat, 88
maximales Paar, 88
Menge aller maximalen Paare, 88
Menge aller maximaler Repeats, 88
Mensch-Maus-Genom, 3
minimal linksnegative Partition, 30
MSS, 1
MSS(a), 11

N

nested suffix, 72

O

offene Lokation, 79
offene Referenz, 79

P

Paar
  exaktes, 85
  maximales, 88
  Tandem-Repeat-, 96
Partition
  fallend rechtsschiefe, 34
  minimal linksnegative, 30
  steigend linksschiefe, 51
primitiv, 97
primitives Tandem-Repeat, 97

R

Rechtsrotation, 97
rechtsschiefe Folge, 34
rechtsschiefer Zeiger, 39
  iterierter, 43
rechtsverzweigend, 96
rechtsverzweigendes Tandem-Repeat-Paar, 96
rechtsverzweigendes Teilwort, 72
Referenz, 61
  offene, 79
Repeat, 85
  exaktes, 85
  k-difference, 85
  k-mismatch, 85
  maximaler, 88
  revers-komplementäres, 95
  Tandem, 96
revers-komplementäres Repeat, 95
revers-komplementäres Wort, 94
rightbranching, 72

S

Σ-Baum, 57
Σ^+-Baum, 57
  kompakter, 58
steigend linksschiefe Partition, 51
Suffix
  aktives, 74
  nested, 72
  verschachteltes, 72
Suffix Tree, 60
Suffix Trie, 59
Suffix-Baum, 60
Suffix-Link, 70

T

Tandem-Arrays, 97
Tandem-Repeat, 96
  Länge, 96
  primitives, 97
Tandem-Repeat-Paar, 96
  Länge, 96
  linksverzweigend, 96
  rechtsverzweigend, 96
Teilwort, 59
  rechtsverzweigendes, 72
Transmembranproteine, 2
Trie, 57
  kompaktifizierter, 58

U

ungapped local alignment, 3

V

verschachteltes Suffix, 72

W

Weighted Maximal Average Scoring Subsequence, 45
WMASS, 45
Wort
  revers-komplementäres, 94
Worttiefe, 58, 98
WOTD-Algorithmus, 67

Z

Zeichen
  komplementäres, 94
  Links-, 90
Zeiger
  iterierter rechtsschiefer, 43
  linksnegativer, 31
  rechtsschiefer, 39