Notizen zu Algorithmen II

Jens Ochsenmeier

6. Februar 2018

Inhaltsverzeichnis

1 Stringology

Inhalt dieses Kapitels:

- Strings sortieren
- Patterns suchen
- Datenkompression

1.1 Strings sortieren

Naive Sortierverfahren, wie sie aus der Vorlesung "Algorithmen 1" bekannt sind, sind beim Sortieren von Strings ineffizient, deswegen gibt es für das Sortieren von Strings andere Algorithmen. Ein solcher ist der **Multikey Quicksort**-Algorithmus:

```
MKQSORT (S: String Seq, l : ℕ): String Seq
assert ∀e, e' ∈ S : e[1...l−1] = e'[1...l−1]
if |S| ≤ 1 then return S
pick p ∈ S randomly
return concatenation of
  MKQSORT (⟨e ∈ S : e[l] < p[l]⟩, l),
  MKQSORT (⟨e ∈ S : e[l] = p[l]⟩, l + 1),
  MKQSORT (⟨e ∈ S : e[l] > p[l]⟩, l)
```

Abbildung 1.1. Pseudocode-Implementierung des Multikey-Quicksort-Algorithmus.

Dieser Algorithmus sortiert eine String-Sequenz und nimmt an, dass die ersten $l-1$ Buchstaben bereits sortiert wurden.
Zuerst wird ein zufälliges Pivotelement gewählt. Danach wird die übergebene Sequenz an Strings wird in drei Teilsequenzen geteilt:

1. Sequenz an Strings, deren l-ter Buchstabe kleiner ist als der l-te Buchstabe des Pivotelements.
2. Sequenz an Strings, deren l-ter Buchstabe derselbe ist wie der l-te Buchstabe des Pivotelements.
3. Sequenz an Strings, deren l-ter Buchstabe größer ist als der l-te Buchstabe des Pivotelements.

Auf die erste und dritte Teilsequenz wird der Algorithmus nun rekursiv mit dem selben Parameter l ausgeführt, da die Buchstaben an der l-ten Position nicht übereinstimmen (müssen) — auf die zweite Teilsequenz wird der Algorithmus rekursiv mit dem Parameter $l+1$ ausgeführt, weil hier die l-ten Buchstaben aller Wörter in der Sequenz gleich sind.
Die Laufzeit des Algorithmus ist in $O(|S| \log |S| + d)$, wobei d die Summe der eindeutigen Präfixe der Strings in S ist.

1.2 Pattern Matching

Hinweis: In diesem Abschnitt sind Arrays 1-basiert.

In diesem Abschnitt wird es darum gehen, alle oder zumindest ein Vorkommen eines **Patterns** $P = p_1 \dots p_m$ in einem gegebenen **Text** $T = t_1 \dots t_n$ zu finden. Im Allgemeinen ist $n \gg m$, also der Text wesentlich länger als das Pattern, das wir in ihm suchen.

Naives Pattern Matching

Das naive Vorgehen ist, an jeder Position von T zu schauen, ob an dieser das gesuchte Pattern vorkommt. Offensichtlich ist dieser Algorithmus in $O(nm)$, da im schlimmsten Fall für jede Position des Textes das gesamte Pattern durchlaufen werden muss. Dieser Algorithmus kann folgendermaßen implementiert werden:

```
NAIVEPATTERNMATCH (P, T)
i, j := 1
while i ≤ n − m + 1
  while j ≤ m ∧ t_{i+j−1} = p_j do j++
  if j > m then return "P occurs at pos i in T"
  i++
  j := 1
```

Abbildung 1.2. Pseudocode-Implementierung des naiven Pattern-Matching-Algorithmus.

Knuth-Morris-Pratt

Ein anderer Algorithmus zum Finden von Patterns in einem gegebenen Text ist der **Knuth-Morris-Pratt-Algorithmus**. Dieser hat sogar optimale Laufzeit, nämlich $O(n + m)$.
Idee dieses Algorithmus ist es, das Pattern eleganter nach vorne zu verschieben, wenn es einen Mismatch zwischen Text und Pattern gibt. Hierfür brauchen wir ein Hilfswerkzeug:
Für einen String S mit Länge k sei $\alpha(S)$ die Länge des Längsten Präfixes von $S_{1\ldots k-1}$, das auch Suffix von $S_{2\ldots k}$ ist.[1]

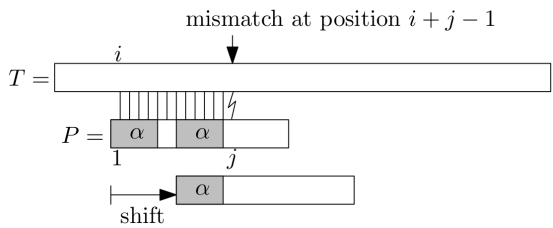


Abbildung 1.3. Idee beim Verschieben des Patterns: α wurde bereits gematcht. Früher als mit dem bereits gematchten Suffix kann das nächste Vorkommen von P nicht auftauchen, also kann mann P direkt um $j - 1 - \alpha$ verschieben.

Der Algorithmus besteht aus zwei Teilen:

1. **Border-Array berechnen** ($O(m)$). Damit die oben erläuterten Verschiebungen nachher effizient durchgeführt werden können, berechnen wir für das leere Wort

[1] Wir lassen absichtlich bei Betrachtung des Präfixes den letzten und bei Betrachtung des Suffixes den ersten Buchstaben weg, damit $\alpha(S) = 0$ ist, wenn $|S| = k = 1$ ist.

und jeden Buchstaben in P einen α-Wert. Diese Werte ergeben das **Border-Array**:

$$\text{border}[j] = \begin{cases} -1, & \text{falls } j = 1 \\ \alpha(P_{1\ldots j-1}), & \text{sonst} \end{cases}.$$

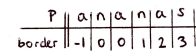


Abbildung 1.4. Beispiel für das Border-Array eines Patterns.

2. **Pattern matchen** ($O(n)$). Nun verwenden wir das erstellte Border-Array, um Vorkommnisse von P in T zu finden. Wir starten sowohl im Text als auch im Pattern an Position 1 und fangen an zu matchen. Kommt es an Position $1 \leq j \leq m$ des Patterns zu einem Mismatch, so können wir P direkt um $j - \text{border}[j] - 1$ verschieben. In Pseudocode sieht das so aus:

```
KMPMatch (P,T)
i, j := 1
while i ≤ n − m + 1
  while j ≤ m ∧ t_{i+j−1} = p_j do j++
  if j > m then return "P occurs at pos i in T "
  i += j − border[j] + 1
  j := max{1, border[j] + 1}
```

Eine Ausführung des Algorithmus kann also folgendermaßen aussehen:

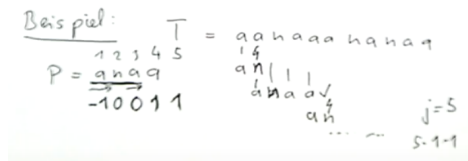


Abbildung 1.5. Beispiel für das Verwenden des Knuth-Morris-Pratt-Algorithmus.

Suffix-Arrays

Im Folgenden werden Arrays wieder mit Position 0 beginnen. Wir verwenden desweiteren folgende Festlegungen:

- Ein **String** ist ein Array von Buchstaben,

$$S[0\ldots n) := S[0\ldots n-1] := [S[0], \ldots, S[n-1]].$$

- Das **Suffix** S_i sei der Substring $S[i\ldots n)$ von S.

- Wir setzen an das Ende jedes Strings ausreichend viele **Endmarkierungen**: $S[n] := S[n+1] := \cdots := 0$. 0 sei per Definition kleiner als alle anderen vorkommenden Zeichen.

Das **Suffix-Array** eines Strings lässt sich nun folgendermaßen konstruieren:

1. Bilde die Menge aller Suffixe S_i ($i = 0, \ldots, n-1$) des Strings.
2. Sortiere die Menge aller Suffixe des Strings (z.B. mit Multikey Quicksort).

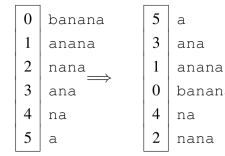


Abbildung 1.6. Beispiel für die Konstruktion des Suffix-Arrays des Strings *"banana"*.

Mithilfe dieses Suffix-Arrays lassen sich später viele Suchprobleme in Linearzeit lösen. Beispielsweise ist die Suche nach dem längsten Substring, der (eventuell mit Überschneidung) zweimal im Text vorkommt, linear — dafür muss nach Berechnung des Suffix-Arrays der längste String gefunden werden, der Präfix von zwei Strings im Suffix-Array ist (im Beispiel oben wäre das *"ana"*).

Berechnung des Suffix-Arrays in Linearzeit

Das Suffix-Array eines Strings lässt sich in Linearzeit berechnen.[2] Hier soll lediglich das Prinzip erläutert werden, genauere Angaben gibt es im Paper.
Wir betrachten den String

$$T[0,n) = \underset{0}{\mathtt{x}}\,\underset{1}{\mathtt{a}}\,\underset{2}{\mathtt{b}}\,\underset{3}{\mathtt{b}}\,\underset{4}{\mathtt{a}}\,\underset{5}{\mathtt{d}}\,\underset{6}{\mathtt{a}}\,\underset{7}{\mathtt{b}}\,\underset{8}{\mathtt{b}}\,\underset{9}{\mathtt{a}}\,\underset{10}{\mathtt{d}}\,\underset{11}{\mathtt{o}}.$$

Unser Ziel ist das Suffix-Array

$$\mathrm{SA} = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0).$$

Wir gehen wie folgt vor:

0. **Suffixe wählen.** Sei

$$B_k = \{i \in [0,n] : i \mod 3 = k\}$$

und $C = B_1 \cup B_2$ sowie S_C die Menge der entsprechenden Suffixe. C ist also die Menge aller Positionen in T, an denen Suffixe mit einer nicht durch 3 teilbaren Länge beginnen. Hier ist $C = \{1, 4, 7, 10, 2, 5, 8, 11\}$.

[2] Kärkkäinen, Sanders, Burkhardt: Linear Work Suffix Array Construction

1. **Gewählte Suffixe sortieren**. Wir fügen am Ende von T beliebig viele $\mathtt{0}$ hinzu und bilden zuerst für $k = 1, 2$ die Strings

   $$R_k = [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}] \cdots [t_{\max B_k} t_{\max B_k+1} t_{\max B_k+2}].$$

   Der Charaktere von R_k sind also Tripel. Das letzte Tripel ist immer eindeutig, weil $t_{\max B_k+2} = 0$. Sei $R = R_1 \odot R_2$.
   Hier ist

   $$R = [\mathtt{abb}][\mathtt{ada}][\mathtt{bba}][\mathtt{do0}][\mathtt{bba}][\mathtt{dab}][\mathtt{bad}][\mathtt{o00}].$$

   Die Ordnung der Suffixe von R stimmt mit der Ordnung der Suffixe S_i überein, deswegen genügt es, die Suffixe von R zu sortieren.
   Wir sortieren R nun, indem wir die einzelnen Charaktere von R sortieren und durch ihren Rang in R ersetzen:

   $$\mathrm{SA}_R = (8, 0, 1, 6, 4, 2, 5, 3, 7).$$

   Nun weisen wir jedem Suffix einen Rang zu. Dazu sei $\mathrm{rank}(S_i)$ der Rang von S_i in C. Für $i \in B_0$ sei $\mathrm{rank}(S_i)$ nicht definiert.
   Hier ist $\mathrm{rank}(S_i) = \bot\ 1\,4\ \bot\ 2\,6\ \bot\ 5\,3\ \bot\ 7\,8\ \bot\ 0\,0$.
2. **Restliche Suffixe sortieren**. Jeder Suffix $S_i \in S_{B_0}$ sei dargestellt durch $(t_i, \mathrm{rank}(S_{i+1}))$. Da wir alle anderen Suffixe oben schon sortiert haben ist $\mathrm{rank}(S_{i+1})$ hier stets definiert.
   Offensichtlich ist

   $$S_i \le S_j \iff (t_i, \mathrm{rank}(S_{i+1})) \le \big(t_j, \mathrm{rank}(S_{j+1})\big),$$

   also lassen sich die Paare Radix-sortieren.
   Hier ist

   $$S_{12} < S_6 < S_9 < S_3 < S_0, \quad \text{weil} \quad (\mathtt{0}, 0) < (\mathtt{a}, 5) < (\mathtt{a}, 7) < (\mathtt{b}, 2) < (\mathtt{x}, 1).$$

3. **Zusammenführen**. Das Zusammenführen erfolgt vergleichsbasiert. Beim Vergleichen on $S_i \in S_C$ mit $S_j \in S_{B_0}$ unterscheiden wir zwei Fälle:

   $$i \in B_1: \quad S_i \le S_j \iff (t_i, \mathrm{rank}(S_{i+1})) \le \big(t_j, \mathrm{rank}(S_{j+1})\big)$$
   $$i \in B_2: \quad S_i \le S_j \iff (t_i, t_{i+1}, \mathrm{rank}(S_{i+2})) \le \big(t_j, t_{j+1}, \mathrm{rank}(S_{j+2})\big)$$

   Hier ist z.B. $S_1 < S_6$ weil $(\mathtt{a}, 4) < (\mathtt{a}, 5)$ und $S_3 < S_8$ weil $(\mathtt{b}, \mathtt{a}, 6) < (\mathtt{b}, \mathtt{a}, 7)$.

Suchen in Suffix-Arrays

Um ein Pattern in einem String zu finden, zu dem man das Suffix-Array konstruiert hat, muss man lediglich ein Suffix finden, dass das gesuchte Pattern als Präfix hat. Man kann so beispielsweise mit binärer Suche in $O(m \log n)$ ein Vorkommen von P in T finden.

Nutzen wir eine zusätzliche Struktur, das **LCP-Array** — dieses speichert in $\text{LCP}[i]$ die Länge des längsten gemeinsamen Präfixes von $\text{SA}[i]$ und $\text{SA}[i-1]$ — so können wir die Suchzeit auf $O(m + \log n)$ reduzieren.

0	banana
1	anana
2	nana
3	ana
4	na
5	a

SA =

5	a
3	ana
1	anana
0	banana
4	na
2	nana

LCP =

0	a
1	**a**na
3	**ana**na
0	banana
0	na
2	**na**na

Abbildung 1.7. Suffixe, Suffix-Array und LCP-Array des Strings *"banana"*.

Um das LCP-Array berechnen zu können brauchen wir das **invertierte Suffix-Array**. Dieses gibt Aufschluss darüber, wo im Suffix-Array ein bestimmter Suffix steht. Offensichtlich ist $\text{SA}^{-1}[\text{SA}[i]] = i$.

Der Algorithmus sieht folgendermaßen aus ($O(n)$):

```
CALCULATELCPARRAY (SA^-1, SA)
h := 0, LCP[1] := 0
for i = 1, ..., n do
  if SA^-1[i] ≠ 1 then
    while t_{i+h} = t_{SA[SA^-1[i]-1]+h} do h++
    LCP[SA^-1[i]] := h
    h := max(0, h − 1)
```

Suffix-Bäume

Noch anschaulicher, allerdings wesentlich platzverbrauchender, sind **Suffix-Bäume** von Strings. Sie sind formal der *kompaktierte Trie der Suffixe* und lassen sich (wenn auch sehr kompliziert) in $O(n)$ berechnen.

Bevor wir den Suffix-Baum eines Strings bilden hängen wir hinten an den String noch einen Charakter dran, der nicht im Alphabet des Strings vorkommt. Das hat den Vorteil, dass anschließend alle Suffixe in einem Blatt des Baums enden.

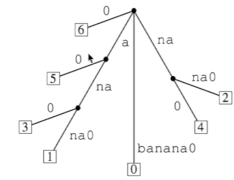


Abbildung 1.8. Beispiel für den Suffix-Baum des Strings *"banana"*.

"Naiv" ist die Erstellung des Suffixbaums in $O(n^2)$. Man kann ihn aber auch aus Suffix-Array und LCP-Array in Linearzeit konstruieren. Dazu hängt man die Suffixe sukzessive in der Tiefe ein, die ihr LCP-Wert angibt:

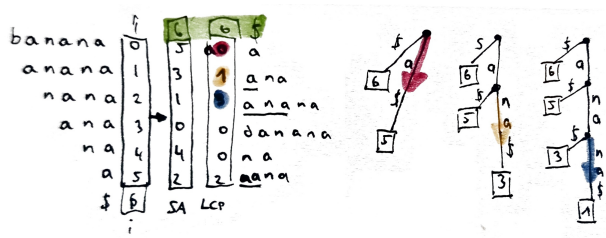


Abbildung 1.9. Sukzessive Konstruktion des Suffix-Baums aus Suffix- und LCP-Array. Zuerst hängt man $ und das erste Suffix (dessen LCP-Wert immer 0 ist) an die Wurzel. Anschließend nutzt man den LCP-Wert des darauffolgenden Suffixes (hier 1), um festzulegen, wo der Suffix zum Baum hinzugefügt werden muss (durch Pfeile gekennzeichnet).

Die Suche in einem Suffix-Baum ist relativ simpel — man muss lediglich den entsprechenden Kanten entlanglaufen, alle Vorkommen des Patterns liegen im entsprechenden Teilbaum.

Zur Angabe der Komplexitäten sind zwei Fälle zu unterscheiden:

1. Die ausgehenden Kanten sind als Arrays der Größe $|\Sigma|$ gespeichert. Dann ist die Suchzeit in $O(m)$ und der Gesamtplatzbedarf in $O(n\,|\Sigma|)$.
2. Die ausgehenden Kanten sind als Arrays gespeichert, deren Größe proportional zur Anzahl der Kinderknoten ist. Dann ist die Suchzeit in $O(m \log |\Sigma|)$ und der Gesamtplatzbedarf in $O(n)$.

1.3 Datenkompression

Eine Anwendung der Suffix-Arrays und -Trees ist die **Datenkompression**. Inhalt dieser Vorlesung wird ausschließlich die *verlustfreie Textkompression* sein.

Wörterbuchbasierte Textkompression

Für besonders große Datenbestände bietet sich eine **wörterbuchbasierte Textkompression** an. Grundidee ist, $\Sigma' \subseteq \Sigma^*$ zu wählen und $S \in \Sigma^*$ durch $S' = \langle s'_1, \ldots, s'_k \rangle \in \Sigma'^*$ zu ersetzen, sodass $S = s'_1 \cdot \cdots \cdot s'_k$ ist. Problem ist der hohe zusätzliche Platzbedarf für das Wörterbuch.

Lempel-Ziv-Kompression

Die **Lempel-Ziv-Kompression** baut das Wörterbuch *on the fly* bei Codierung und Decodierung, sodass dieses nicht explizit gespeichert werden muss.

```
NAIVELZCOMPRESS(⟨s_1, ..., s_n⟩, Σ)
D := Σ    // init dictionary
p := s_1    // current string
for i := 2 to n do
  if p · s_i ∈ D then p := p · s_i
  else
    output code for p
    D := D ∪ p · s_i
    p := s_i
output code for p
```

```
NAIVELZDECODE(⟨c_1, ..., c_k⟩)
D := Σ
output decode(c_1)
for i := 2 to k do
  if c_i ∈ D then
    D := D ∪ decode(c_{i-1}) · decode(c_i)[1]
  else
    D := D ∪ decode(c_{i-1}) · decode(c_{i-1})[1]
  output decode(c_i)
```

Abbildung 1.10. Kompressions- und Dekodierungs-Algorithmus für die Lempel-Ziv-Kompression.

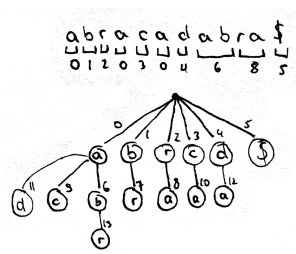


Abbildung 1.11. Beispiel für das Bilden der Lempel-Ziv-Kompression des Wortes *"abracadabra"*. Der Baum wird *on the fly* berechnet und nicht übergeben, sondern nur die komprimierte Information und das Alphabet.

2

Range Minimum Queries

Eine **range minimum Query** gibt für ein array A ($|A| =: n$) die Position des kleinsten Elements zwischen zwei Begrenzern $1 \leq \boldsymbol{l} < \boldsymbol{r} \leq n$ zurück:

$$\mathrm{rmq}_A(l, r) = (\arg) \min_{l \leq k \leq r} A[k]$$

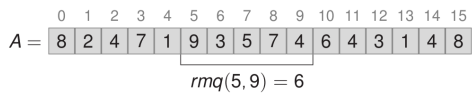


Abbildung 2.1. Beispiel einer range minimum query.

Ziel dieses Kapitels wird sein, einen Algorithmus anzugeben, mit dem eine RMQ-Abfrage in konstanter Zeit beantwortet werden kann, nachdem eine $2n + o(n)$ große Datenstruktur in Linearzeit vorbereitet wurde.

Ein naiver Ansatz, um eine range minimum query auszuführen, ist, einfach das Array zu durchlaufen und das Minimum zu speichern (und wenn nötig zu aktualisieren). Dafür ist keine Vorbereitungsarbeit nötig (also $O(1)$) und die Abfrage ist in $O(n)$. Wir notieren

$$\langle O(1), O(n) \rangle .$$

2.1 Lösung 1 — $\langle O(n), O(\log n) \rangle$

Baut man einen binären Suchbaum über das Array auf, so lässt sich die Komplexität der Abfrage auf $O(\log n)$ reduzieren.

Hierzu betrachtet man die größtmöglichen Knoten, die vollständig im Abfrageintervall liegen (grün dargestellt), und berechnet das Minimum dieser.

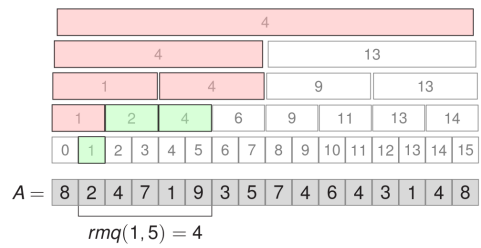


Abbildung 2.2. Die grauen Felder stellen die Knoten des binären Suchbaums dar, sie beinhalten die Position des Arrays, an der der Teilbaum, dessen Wurzel sie sind, den minimalen Wert annimmt. Die größten Knoten, die vollständig im Intervall liegen, sind grün markiert..

2.2 Lösung 2 — $\langle O(n \log n), O(1) \rangle$

Wir reduzieren nun die Zeit, die zum Bearbeiten der rmq benötigt wird, auf $O(1)$, indem wir für jedes $A[i]$ ein Array $M_i[0, \log n]$ vorberechnen. Es sei

$$M_i[j] = \text{rmq}_A(i, i + 2^j - 1).$$

Idee ist es nun, $\text{rmq}_A(l, r)$ aus der Überdeckung des Intervalls durch zwei Zweierpotenzen zu berechnen.

Wir suchen dafür $2^{\lfloor l-r \rfloor}$, also die größte Zweierpotenz, die kleiner ist als die Länge des Intervalls. Offensichtlich ist diese Zweierpotenz mehr als halb so groß wie das Intervall, also ist $\text{rmq}_A(l, r)$ entweder das Minimum der ersten oder zweiten überdeckenden Zweierpotenz.

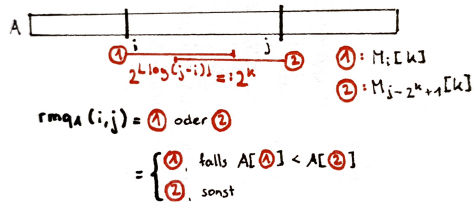


Abbildung 2.3. Funktionsweise des zweiten rmq-Algorithmus.

2.3 Lösung 3 — $\langle O(n \log \log n), O(1) \rangle$

Wir wenden folgende Prozedur an:

1. A in $t = \frac{n}{\log n}$ Blöcke $B_0, \ldots, B_{t-1}$ der Größe $\log n$ unterteilen.
2. Array $S[0, t-1]$ mit $S[i] = \min\{x \in B_i\}$ erstellen, rmq-Struktur nach Lösung 2 für S berechnen.
3. Für jeden Block B_i rmq-Struktur nach Lösung 2 berechnen.

Diese Prozedur liegt in $O(n \log \log n)$.
Soll nun $\text{rmq}_A(l, r)$ bestimmt werden, so geht das folgendermaßen:

1. Bestimme die Blöcke $l \in B_{l'}$ und $r \in B_{r'}$.
2. Berechne $m = \text{rmq}_S(l' + 1, r' - 1)$. Wir nutzen also die Struktur über S, um die rmq-Werte der Blöcke zwischen den beiden Grenzblöcken zu berechnen.
3. Es seien k_0, k_1, k_2 die rmq_A-Resultate in den Blöcken l', r' und m
4. $\text{rmq}_A(l, r) = \arg\min\{A[k_0], A[k_1], A[k_2]\}$.

2.4 Lösung 4 — $\langle O(n), O(1) \rangle$

Zur Konstruktion des Algorithmus mit linearem Platzverbrauch benötigen wir kartesische Bäume.

Kartesischer Baum

Der **kartesische Baum** eines Arrays A ist folgendermaßen definiert:

- *Wurzel* des Baums ist das (linkste) kleinste Element m des Arrays, mit Position als Label.

- Die Wurzel hat zwei Kindknoten — die Wurzel des linken und die Wurzel des rechten Subarrays von A (jeweils von m aus).
- Rekursion.

Implementierung

Die Implementierung funktioniert so:

1. Partitioniere das Array in Blöcke der Größe s — jeder Block entspricht einem kartesischen Baum.
2. Berechne alle s^2 Möglichkeiten aller $\frac{1}{s+1}\binom{2s}{s}$ möglichen kartesischen Bäume der Größe s in einer Tabelle P. Diese benötigt $O(2^{2s}s^2)$ Speicher, also braucht P für $s = \frac{\log n}{4}$ nur $o(n)$ Speicher.
3. Berechne nach Lösung 2 die Struktur für das Array A', welche aus den blockweisen Minima von A besteht ($O(n)$ Zeit, $O(n)$ Platz).

2.5 Lowest Common Ancestor

Gegeben sei ein Baum T mit Wurzel, $|T| = n$. Es sei

$$\mathrm{LCA}_T(v, w)$$

der **lowest common Ancestor** der Knoten v, w in T, also der Knoten $a \in T$, der

- Vorgänger von v und w ist und
- maximalen Abstand zur Wurzel hat.

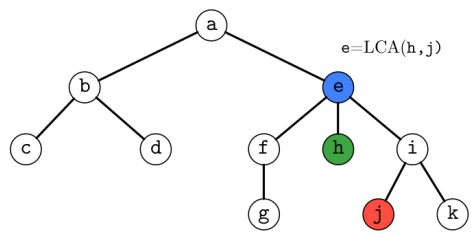


Abbildung 2.4. lowest common ancestor der Knoten h und j ist e.

Von RMQ zu LCA

Lemma 2.5.1. Gibt es eine $\langle f(n), g(n)\rangle$-Lösung für RMQ, so gibt es eine Lösung in

$$\langle f(2n-1)+O(n), g(2n-1)+O(1)\rangle$$

für LCA.

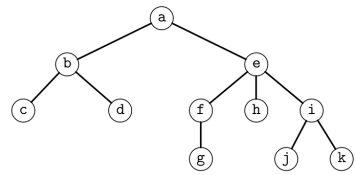
Beweis. Sei

- T der kartesische Baum des Arrays A,
- $E[0,\ldots,2n-2]$ das Knoten-Array, die in einer DFS-Euler-Tour von T besucht wurden,
- $L[0,\ldots,2n-2]$ die entsprechende Tiefe der Knoten in E,
- $R[0,\ldots,n-1]$ ein Array mit $R[i] = \min\{j : E[j] = i\}$ für jeden Knoten $i \in T$, also wo in der Euler-Tour der Knoten i das erste Mal auftaucht.

Dann ist

$$\mathrm{LCA}_T(v,w) = E[\mathrm{RMQ}_L(\min\{R[v],R[w]\}, \max\{R[v],R[w]\})].$$

□




```
i = 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
E = a b c b d b a e f g f e h e i j i k i e a
L = 0 1 2 1 2 1 0 1 2 3 2 1 2 1 2 3 2 3 2 1 0

    a b c d e f g  h  i  j  k
R = 0 1 2 4 7 8 9 12 14 15 17
```

Abbildung 2.5. Konstruktion der Hilfsarrays für LCA.

Achtung: Die Tiefe zweier Knoten, die nacheinander bei der Euler-Tour besucht wurden, kann sich höchstens um 1 unterscheiden, also

$$(L[i] - L[i+1]) \in \{-1, 1\}.$$

Wir müssen also nur RMQs über Arrays lösen, bei denen sich zwei nacheinanderfolgende Elemente um 1 unterscheiden. Das ist das sogenannte **±1-RMQ**.

Wir werden nun im Folgenden ausnutzen, dass wir zum Lösen des LCA-Problems nur ±1-RMQs betrachten müssen.

LCA in $\langle O(n), O(1)\rangle$ auf $4n + o(n)$ Bits

Wir verwenden folgende Konstuktion:

- **Kartesischen Baum bauen**. Wir gehen das Array A von links nach rechts durch und fügen die Elemente sukzessive dem Baum hinzu. Dabei sind stets die Regeln für einen kartesischen Baum einzuhalten, also
  - *Wurzel* des Baums ist das (linkste) kleinste Element m des Arrays.
  - Die Wurzel hat zwei Kindknoten — die Wurzel des linken und die Wurzel des rechten Subarrays von A (jeweils von m aus).
  - Rekursion.
- **Dummy-Blätter hinzufügen**. An jeden Knoten des Baums hängen wir ein Dummy-Blatt.
- **DFS**. Wir konstruieren eine Sequenz an Klammern, indem wir ein DFS durchführen. Traversieren wir einen Knoten das erste Mal, so setzen wir "(", beim zweiten Mal ")". In einem Blatt drehen wir um und setzen daher direkt beide Klammern.

  Wir können die Klammernfolge in $4n$ Bits speichern, indem wir "(" durch `1` und ")" durch `0` repräsentieren.

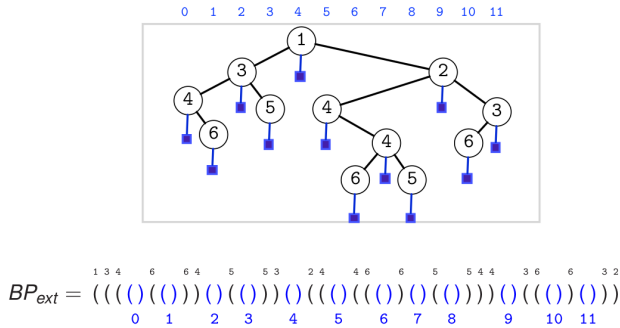


Abbildung 2.6. Konstruktion des kartesischen Baums mit Dummy-Blättern (blau) für das Array "463514645236". Anschließend wird die Klammernfolge gebildet. In blau hervorgehoben sind die "Wendepunkte", also die Dummy-Blätter, und ihr Index.

- **RMQ berechnen**. Zur Berechnung von $\text{rmq}_A(l, r)$ brauchen wir drei Hilfsfunktionen:
  - $\text{rank}(\texttt{pos}, \texttt{char}, \text{BP}_{\text{ext}})$ gibt die Anzahl an Vorkommisse von `char` (z.B. ")") bis `pos` in BP_{ext} an.
  - $\text{excess}(i) = \text{rank}(i+1, (, \text{BP}_{\text{ext}}) - \text{rank}(i+1,), \text{BP}_{\text{ext}})$
  - $\text{select}(j, \texttt{char}, \text{BP}_{\text{ext}})$ gibt die Position des j-ten Vorkommens von `char` in BP_{ext} an.

Wir können nun den RMQ-Wert folgendermaßen berechnen:

```
RMQ (A,l,r)
lpos := select(l + 1,(),BP_ext)
rpos := select(r + 1,(),BP_ext)
return rank(rmq^±1_excess(lpos,rpos + 1),(),BP_ext)
```

LCA in $\langle O(n), O(1)\rangle$ auf $2n + o(n)$ Bits

Die Anzahl an benötigten Bits lässt sich im Vergleich zum vorhergehenden Ansatz noch weiter verkleinern. Wir transformieren dazu den kartesischen Baum — der ja ein Binärbaum ist — in einen allgemeinen Baum. Dazu gehen wir folgendermaßen vor:

1. Füge einen Elternknoten zur Wurzel des Baums hinzu (der hinzugefügte Knoten ist also die neue Wurzel).
2. Nehme von jedem Knoten v — von der neuen Wurzel ausgehend — den rechten Kindknoten w, und füge alle Knoten auf dem linkesten Pfad von w ausgehend zu den Kindknoten von v hinzu.

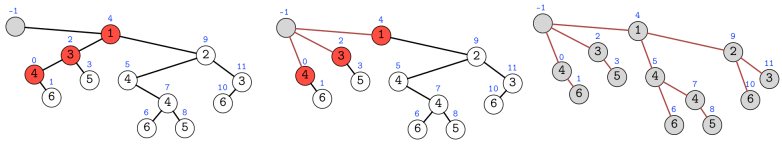


Abbildung 2.7. Kartesischer Baum mit neuer Wurzel. Die roten Knoten in der ersten Grafik sind die Knoten, die zu v — hier die neue Wurzel — hinzugefügt werden. Durch Hinzufügen dieser Knoten zu den Kindknoten von v hat der Baum die Struktur in Grafik 2. Nun wird dieser Prozess rekursiv auf die drei roten Knoten ausgeführt. Ist die Rekursion vollständig abgeschlossen sieht der Baum wie in der dritten Grafik aus.

Diese Transformation lässt sich bei Bedarf auch rückgängig machen; man kann also, wenn nötig, wieder den Binärbaum konstruieren.
Nun lässt sich wieder die Klammernreihe BP von oben bauen. Mit dieser können wir nun RMQ-Abfragen lösen:

```
RMQ (A,l,r)
lpos := select(l + 2,(,BP)
rpos := select(r + 2,(,BP)
return rank(rmq^±1_excess(lpos − 1,rpos),(,BP) − 1
```

3

Burrows-Wheeler-Transformation

Die **Burrows-Wheeler-Transformation** erzeugt eine sinnvolle Permutation des eingegebenen Strings; sie gruppiert Zeichen mit ähnlichem Kontext nahe beieinander. Die Struktur der Permutation beinhaltet alle Informationen, die benötigt werden, um eine Rücktransformation durchzuführen, es sind also keine Zusatzinformationen nötig. Hin- und Rücktransformation geht in $O(n)$. Sie wird hauptsächlich zur Vorverarbeitung statischer Texte genutzt, um sie komprimieren, indizieren und in ihnen suchen zu können.

3.1 Konstruktion

Sei T = `lalalangng$` der gegebene String (mit angehängtem $-Zeichen), $n = |T|$ und $T^{(i)}$ die i-te Permutation von T (durch i mal den vordersten Buchstaben nehmen und hinten anhängen). Man erhält die Burrows-Wheeler-Transformation von T so:

1. Schreibe $T^{(1)}$ bis $T^{(n)}$ untereinander.
2. Sortiere $T^{(1)}$ bis $T^{(n)}$.
3. Die letzte Spalte ist T^{BWT} (= L), die Burrows-Wheeler-Transformation von T.

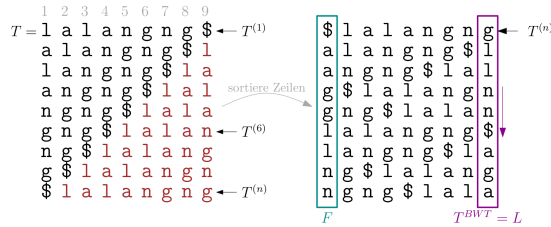


Abbildung 3.1. Konstruktion der Burrows-Wheler-Transformation von T = `lalalangng$`, T^{BWT} = `gllnn$ aga`. Da T^{BWT} die letzte Spalte ist schreibt man oft auch L stattdessen. Die erste Spalte wird auch F genannt.

Naiv benötigt die Berechnung von T^{BWT} $O(n^2 + n \log n)$ Schritte. Die Berechnungszeit lässt sich aber auf $O(n)$ reduzieren.

3.2 Beobachtungen

Folgende Eigenschaften lassen sich feststellen:

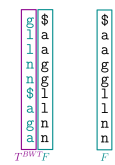
- Die Zeilen der oben konstruierten Matrix enthalten die sortierten Suffixe von T (vom Zeilenstart bis $ gehend).
- Die Zeichen der letzten Spalte (also T^{BWT}) sind also die Zeichen, die vor dem zu ihrer Zeile gehörenden Suffix stehen. Formaler ist $T^{\text{BWT}}[i]$ das Zeichen vor dem i-ten Suffix in T:

  $$T^{\text{BWT}}[i] = L[i] = T[\text{SA}[i] - 1] = T^{(\text{SA}[i])}[n]$$

  Da wir mithilfe des DC3-Algorithmus das Suffix-Array in Linearzeit berechnen können, können wir auch die Burrows-Wheeler-Transformation in Linearzeit bestimmen.

3.3 Rücktransformation

Wir können aus einer vorliegenden T^{BWT} einfach F — also die erste Spalte der Matrix — konstruieren, indem wir die Buchstaben von T^{BWT} sortieren. Hängen wir nun T^{BWT} und F hintereinander, so haben wir bereits Buchstabenpaare, die so auch in T auftreten. Sortieren wir nun die beiden Spalten (also die Buchstabenpaare) lexikographisch, so erhalten wir die auf F folgende Spalte. Durch diesen Prozess lässt sich die gesamte Matrix und somit T rekonstruieren.



Diese Art der Rücktransformation benötigt $O(n^2 \log n)$ Schritte. Im Folgenden werden wir die Rücktransformation auf Linearzeit reduzieren. Dazu benötigen wir **Last-to-front mapping**:

$\mathrm{LF}[i] :=$ Position in L, an der Vorgänger von $L[i]$ steht

Da die Spalten der BWT-Matrix zyklisch sind, ist der Vorgänger von $L[i]$ derjenige Buchstabe, der in $F[i]$ steht, also

$\mathrm{LF}[i] =$ Position, an der $L[i]$ in F steht

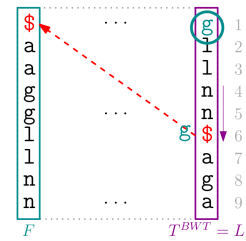


Abbildung 3.2. Gesucht ist der Vorgänger von \$. Stellen wir uns T^{BWT} ein zweites Mal links von F vor, so sehen wir, dass es g ist.

Wir erhalten folgenden Zusammenhang:

$$\mathrm{LF}[i] = j \Leftrightarrow T^{(\mathrm{SA}[j])} = \left(T^{(\mathrm{SA}[i])}\right)^{(n)}.$$

Weitere Überlegungen zur Rücktransformation

Wir können desweiteren folgende Beobachtungen an T^{BWT} machen:

- Gleiche Zeichen haben gleiche Reihenfolge in F und L.
- Falls $L[i] = L[j]$ für $i < j$, dann ist $\mathrm{LF}[i] < \mathrm{LF}[j]$.

Grund dafür ist, dass die Zeilen der BWT-Matrix lexikographisch sortiert sind.

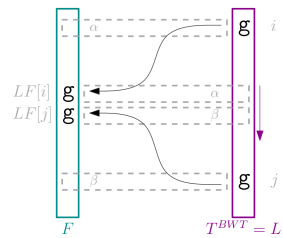


Abbildung 3.3. Präfixe α und β und wie sie in der Matrix vorkommen.

Wir können also LF rein aus T^{BWT} berechnen. Dazu brauchen wir nur zwei Hilfsfunktionen:

- $C(a) :=$ # Zeichen $< a$
- $\mathrm{occ}[i] :=$ # Zeichen $= L[i]$ in $L[1 \ldots i]$

Nun können wir LF$[i]$ darstellen als

$$\text{LF}[i] = C(L[i]) + \text{occ}[i]$$

und können somit LF in $O(n)$ berechnen, da sich C und occ in Linearzeit berechnen lassen.

Implementierung

Zuerst berechnen wir LF. Hier sieht die Implementierung so aus:

1. Initialisiere occ und h. h sei ein Array, das zählt, wie oft ein bestimmter Buchstabe vorkommt, damit wir nachher C gescheit berechnen können.
2. Laufe durch $L = T^{\text{BWT}}$ $(i = 1 \dots n)$
   - $h(L[i])$++
   - $\text{occ}(L[i]) = h(L[i])$
3. Konstruiere C aus h: $C(\$) = 0$, $C(\alpha) = C(\alpha - 1) + h(\alpha - 1)$ (α ist ein Buchstabe, $\alpha - 1$ sein Vorgänger)
4. LF$[i] = C(L[i]) + \text{occ}[i]$

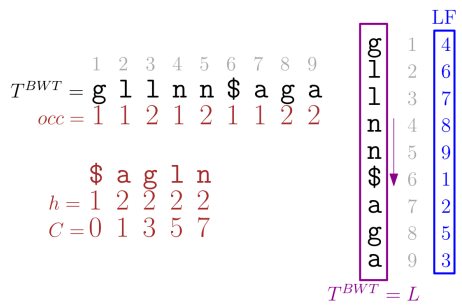
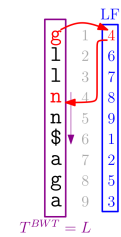


Abbildung 3.4. Beispiel des Algorithmus zur Berechnung von LF nach Durchführung.

Nun kann T von rechts nach links berechnet werden:

1. $T[n] = \$ \Rightarrow \text{LF}[\cdot] = 1$. Das ist unabhängig von T so.
2. $L[1] = \text{g} \Rightarrow T[n-1] = \text{g} \Rightarrow \text{LF}[1] = 4$
3. $L[4] = \text{n} \Rightarrow T[n-2] = \text{n} \Rightarrow \cdots$

Also geht auch die Rücktransformation in $O(n)$.



3.4 Was bringt die BWT?

Die Vorteile der Burrows-Wheeler-Transformation sind nicht direkt erkennbar — sie nutzt dieselben Zeichen wie T und benötigt den gleichen Platz.
Allerdings wird die *Komprimierung stark vereinfacht*, weil Zeichen mit ähnlichem Kontext gruppiert werden. Besonders gut funktioniert sie auf Texten mit vielen gleichen Substrings, wie beispielsweise einem englischen Fließtext. Zur Vereinfachung von *Indexierung* und *Suche* steuert sie auch bei, weil Vorgänger von Suffixen einfach bestimmt werden können.
Im Folgenden werden wir uns die Burrows-Wheeler-Transformation im Kontext von *Kompression* und *Suche* anschauen.

3.5 Kompression

Wir schauen uns zwei Kompressionsmöglichkeiten an: die *move to front*-Kodierung und die Huffman-Kodierung

MTF-Kodierung

Idee der **MTF-Kodierung** ist es, lokale Redundanz zu nutzen und so kleine Zahlen für gleiche Zeichen, die nahe beieinander liegen, zu verwenden. Die Umsetzung funktioniert so:

1. Initialisiere Y mit Alphabet von T^{BWT}.
2. Durchlaufe T^{BWT} $(i = 1, \ldots, n)$
   - Generiere $R[1, \ldots, n]$, wobei $R[i]$ die Position von $T^{\text{BWT}}[i]$ in Y codiert.
   - Schiebe $T^{\text{BWT}}[i]$ an den Anfang von Y.

$$T^{BWT} = \begin{array}{c} \scriptstyle 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ \ 9 \\ \texttt{g}\ \texttt{l}\ \texttt{l}\ \texttt{n}\ \texttt{n}\ \texttt{a}\ \texttt{g}\ \texttt{a}\ \texttt{\$} \end{array} \qquad R = \begin{array}{c} \\ \texttt{3}\ \texttt{4} \end{array}$$

$$Y = \begin{array}{c} \scriptstyle 1\ \ 2\ \ 3\ \ 4\ \ 5 \\ \texttt{\$}\ \texttt{a}\ \texttt{g}\ \texttt{l}\ \texttt{n} \\ \texttt{g}\ \texttt{\$}\ \texttt{a}\ \texttt{l}\ \texttt{n} \\ \texttt{l}\ \texttt{g}\ \texttt{\$}\ \texttt{a}\ \texttt{n} \end{array}$$

Abbildung 3.5. Es wurde hier gerade die 4 eingefügt und deswegen 1 in Y nach vorne genommen. Als nächstes muss 1 codiert (≙ 1) und Y anschließend nicht verändert werden, weil 1 ja eh schon ganz vorne steht.

Huffman-Kodierung

Die **Huffman-Kodierung** erzeugt präfixfreie Codes variabler Länge. Der Ablauf ist:

1. Notiere vorkommende Symbole und ihre jeweiligen Häufigkeiten. Sie sind die Blätter des (binären) Huffman-Baumes.
2. Verknüpfe die zwei seltenstem Knoten in einem neuen Knoten. Die Häufigkeit des neuen Knotens ist die Summe der Häufigkeiten seiner Kinder. Dies erfolgt nun iterativ.
3. Die Wurzel hat relative Häufigkeit 1 (bzw. absolute Häufigkeit $|T|$).
4. Beschrifte die Kanten zwischen einem Knoten und seinen beiden Kindern mit 0 und 1. Der Pfad von der Wurzel zu einem bestimmten Blatt ergibt den Code des Symbols, zu dem das Blatt gehört.

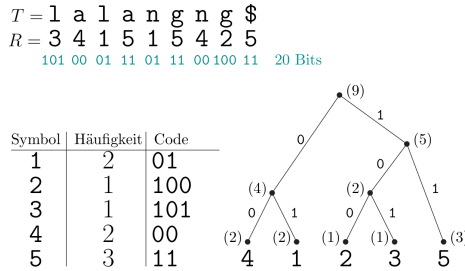


Abbildung 3.6. MTF-Kodierung R von T, die Häufigkeit der in R vorkommenden Symbole und die mit dem Huffman-Baum erzeugten Codes.

3.6 Suche in der Burrows-Wheeler-Transformation

Wir möchten nun in T^{BWT} nach einem Pattern P suchen. Hier sei

P = `bar` und

T = `abracadabrabarbara$`.

Wir benötigen dazu zwei Hilfsmittel:

- Das Array C beinhalte für jeden eindeutigen Buchstaben in $t \in T$ die Position des ersten Suffixes im Suffix-Array, das mit t beginnt.
- $\text{rank}(i, X, \text{BWT})$ gibt an, wie oft ein Buchstabe X in $\text{BWT}[0, \ldots, i-1]$ vorkommt.