Algorithmische Bioinformatik

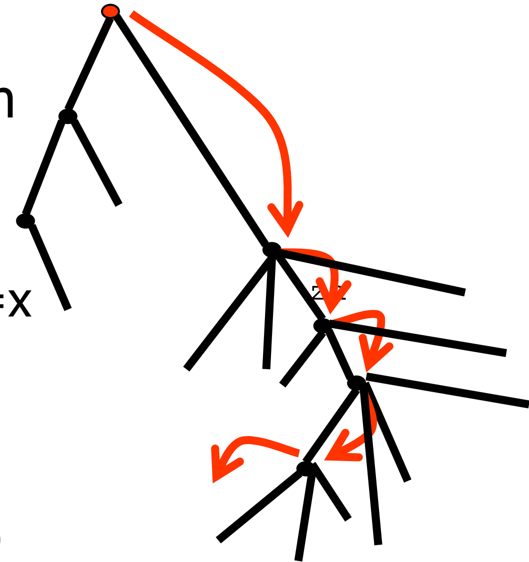
Suffixarrays

Ulf Leser

Wissensmanagement in der Bioinformatik

Suche in Suffixbäumen

- Matche Suchstring bis Erfolg oder Mismatch
  - An jedem Knoten Entscheidung basierend auf erstem Zeichen des Kantenlabels
  - Kantenlabel beginnen mit verschiedenen Zeichen
- Wie trifft man diese Entscheidung?
  - Array in der Größe des Alphabets Σ
    - Zelle x mit Pointer auf Kante k, wenn label(k)[1]=x
    - Konstante Zeit pro Knoten
  - Verkettete Liste
    - Pointer auf Kinderkanten sind verkettet
    - Reihenfolge der Kinder wie Einfügung (unsortiert)
    - Lineare (in |Σ|) Zeit pro Knoten
  - Wachsendes, sortiertes Array
    - Pointer auf Kinderkanten sind alphabetisch sortiert und direkter Zugriff auf Pointer
    - Mit binärer Suche: log(|Σ|)



Konstruktion mit Ukkonen's Algorithmus

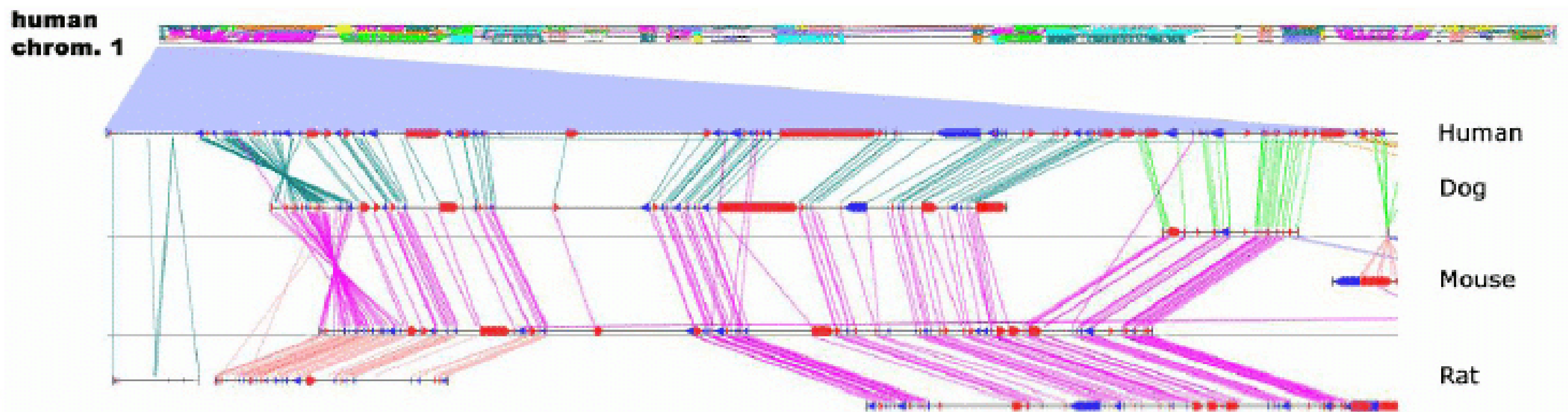
- Baum wird beim Konstruieren auf zwei Arten traversiert
  - Verfolgen von Suffix-Links – Sprünge in andere Teilbäume
  - Knotenhüpfen – Abstieg in einen Teilbaum
- Kein Problem beim Aufbau im Hauptspeicher, aber ...

Bisher bestes Verfahren

- TDD – Top-Down, Disc-Based
  - Tata, Hankins, Patel: „Practical Suffix Tree Construction“, VLDB 2004
  - Partitionierung ähnlich Hunt
    - Sequenz wird gecached – Vermeidung der k Scans
  - In einer Partition
    - Alle Suffixe bestimmen
    - Sortieren (man braucht nur Pointer in die Originalsequenz)
    - In sortierter Reihenfolge Suffixtree bauen
      - Mit einigen Tricks kriegt man O(n*log(n)) Average Case hin
  - Welche Datenstruktur bekommt wie viel Speicher?
    - Sequenz, sortierte Liste, wachsender Partitionsbaum
- Ergebnis: 30h für 3GB
- Auch im Main-Memory besser als Ukkonnen
  - Failure Links verursachen Cache Misses im CPU Cache
- Man kennt auch schon garantiert lineare Secondary-Memory Algorithmen, sind aber wohl langsamer als TDD

Whole Genome Aligment

- Genome unterliegen Evolution
- Neben Punktmutationen treten großräumige Veränderungen auf
  - Duplikation von Chromosomen
  - Duplikation von langen DNA Abschnitten
- Duplikate unterliegen dann unabhängiger Evolution
- In nahe verwandten Organismen (z.B. Säugetiere) sind Genome „durcheinander geschüttelte", approximative Kopien voneinander



Finden von MUMs

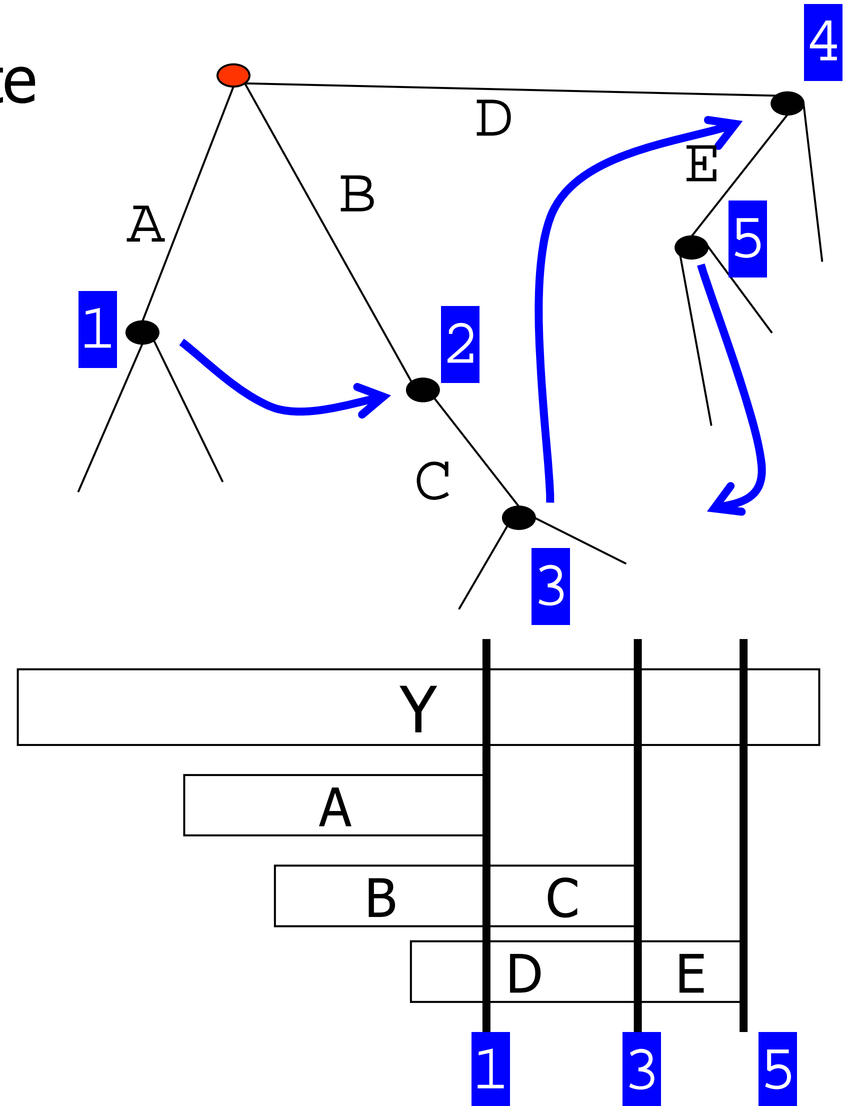
- Beobachtung
  - Evolutionärer Abstand in nahe verwandten Spezies ist klein
  - Trotz unabhängiger Evolution findet man genügend lange, identische Sequenzabschnitte
  - Also: Anwendung für exaktes Stringmatching
- Aufgabe
  - Gegeben zwei Chromosome X, Y (Mensch und Maus)
  - Finde Bereiche von X, die identisch in Y vorkommen
- Genauer: Finde längste identische Teilsequenzen über einer Mindestlänge
- Außerdem: Um klare Zuordnungen zu ermöglichen, beschränkt man sich auf eindeutige Teilsequenzen
  - Dürfen in X und Y nur jeweils einmal vorkommen
- Wir wollen alle MUMs zwischen Maus und Menschen finden
  - Vorschläge?

Möglichkeit 2

- Idee nach Chang and Lawler; verwendet in MUMmer 2
  - Wir bauen Suffixtrees nur für ein Chromosom X
  - Suffix-Links werden behalten
  - Alle Chromosomen der anderen Spezies „streamen" wir gegen X
- Komplexität
  - Man baut 2c „kleinere" Suffixbäume in jeweils O(m)
  - Gegen jeden streamen wir c mal in O(m)
  - Zusammen: $O(2c*m) + O(c^2*m)$
- Hauptvorteil: IO vernachlässigbar
  - Suffixbäume kann man im Hauptspeicher bauen
  - Vergleichssequenz wird sequentiell gelesen

Illustration

- Vereinfachte Annahme: Letzte Matches immer in Knoten
- Mismatch in Knoten 1
  - mit Label A
- SL zu Knoten 2
  - mit Label B
  - B ist Suffix von A
- Weiter in X matchen bis Mismatch in Knoten 3
  - mit Label BC
- SL zu 4
  - mit Label D
  - D ist Suffix BC
- Weiter in Y matchen bis …
- …



Fast richtig

- Wenn man ab k` weiter matched
  - Und einen MUM findet
  - Dann ist der nicht notwendigerweise maximal
  - Das rechte Ende ist klar
    - Der aktuelle Mismatch
  - Aber das linke Ende nicht
- Linke Enden müssen explizit geprüft werden
  - In Y und in T vor dem Suffix nach links vergleichen bis Mismatch
- Damit matched man Zeichen in Y mehrmals
  - Außerdem matched man die Zeichen zwischen k und einem Mismatch zweimal
- Aber: Wenn minimale MUM Länge sinnvoll groß, gibt es nur wenige MUMs – praktisch lineare Laufzeit (in |Y|)

Inhalt dieser Vorlesung

- Suffixarrays
- Suche
- Konstruktionsalgorithmus von Manber / Myers

Motivation

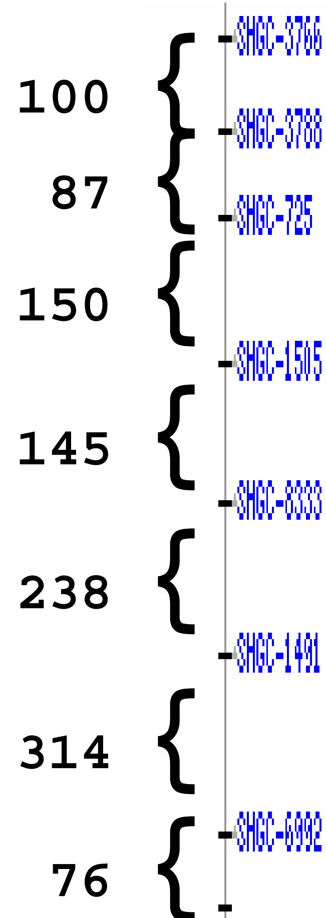
- Performance (Zeit / Speicherverbrauch) von Suffixarrays praktisch unabhängig von der Größe des Alphabets
- Warum ist das wichtig?
  - DNA/RNA: $|\Sigma|=4$
    - Platzbedarf der Suffixbäume vertretbar
  - Proteine: $|\Sigma|=20$
    - Anwendungsabhängige Entscheidung
  - Chinesisch: ~ 2000 Zeichen
- Aber es gibt auch ganz andere „Alphabete"
  - Suffixbäume/arrays haben so schnelle Algorithmen, dass es sich oftmals lohnt, ein Problem als String zu modellieren
  - Beispiel: Restriktionskarten

Vergleich von Genomkarten

- Situation
  - Chromosom X ist sequenziert
  - Schnittstellen des Restriktionsenzyms Y auf X bekannt
  - Labor unternimmt „Position Cloning" auf der Suche nach einem Gen
  - Ergebnis ist ein Clone Z, der das Gen enthält
  - Frage: Liegt dieser Clone auf Chromosome X? Wo?
- Sehr billige Möglichkeit
  - [Electronic PCR oder …]
  - Schnittstellen von Y auf Z bestimmen
  - Mit den Schnittstellen auf X vergleichen
  - Das Muster der Stücklängen von Z muss irgendwo in X liegen

Modellierung als Stringmatching

- Restriktionskarten als Strings
  - Wichtig sind die Abstände der Schnittpunkte (also die Länge der Fragmente)
- Chromosome X
  - X�=„100,87,150,145,238,314,76,...“
- Clone Z
  - Z‘=„145,238,314,76,...“
- ➢ Z liegt auf X wenn Z‘ $\subseteq$ X‘
- Wie groß ist hier das Alphabet?
  - Sehr groß
  - Abhängig vom Enzym (Häufigkeit der Bindungsstelle)



Einschub

- Stimmt nicht ganz
- Wir können die Strings auch in Dezimaldarstellung vergleichen
  - „145,238,314,76,...“ $\subseteq$ „100,87,150,145,238,314,76,...“
  - Oder Codierung nach Ordnung: „4,6,7,1“ $\subseteq$ „3,2,5,4,6,7,1 “
- Man kann auch alles binär kodieren: $|\Sigma|=2$
- Preis: Vergrößerung des Strings
- Trade-Off

Suffixarrays

- Definition
  - *Das Suffixarray A für String S ist ein Integerarray der Länge |S|, in dem A[i] die Startposition des i-ten Suffix von S, sortiert nach lexikographischer Ordnung, enthält*
- Beispiel

```
12345678901
mississippi
```

mississippi	i	A[1]=11
ississippi	ippi	A[2]=8
ssissippi	issippi	A[3]=5
sissippi	ississippi	A[4]=2
issippi	mississippi	A[5]=1
ssippi	pi	A[6]=10
sippi	ppi	A[7]=9
ippi	sippi	A[8]=7
ppi	sissippi	A[9]=4
pi	ssippi	A[10]=6
i	ssissippi	A[11]=3

Konstruktion von Suffixarrays

- Behauptung
  - Suche nach P in A benötigt nur O(n+log(m))
  - Wie? Später
- Zunächst: Konstruktion eines Suffixarrays
  - In linearer Zeit aus Suffixbaum
  - Erinnerung: Im Suffixbaum gibt es ein Blatt pro Suffix
  - Wir suchen die Blätter – und zwar gleich in der richtigen Reihenfolge – und füllen dabei das Array von vorne nach hinten
    - Suffixe in beliebiger Reihenfolge aufzählen und dann sortieren wäre schlecht
    - Beobachtung: Suffixbäume sind ja in gewisser Weise „sortiert"

Trick zur linearen Konstruktion

- Ablauf in „lexikographischer" Depth-First Ordnung
  - Wir laufen den Suffixbaum Depth-First ab
  - An jedem Knoten wählen wir die Kinder in der lexikographischen Reihenfolge der Kantenlabel
    - „$" gilt als kleiner als alle anderen Zeichen
  - Reihenfolge der Blätter bildet das Array
    - Erstes Blatt mit Beschriftung $i_1 \Rightarrow A[1] = i_1$
    - Zweites Blatt mit Beschriftung $i_2 \Rightarrow A[2] = i_2$
- Komplexität: O(m)
  - Depth-First ist abhängig von Anzahl Knoten
  - Wenn die Kinder als sortierte verkettete Liste oder als Array gespeichert sind ist jede Entscheidung konstant
- Aber
  - Wir haben unser Platzproblem nicht gelöst
  - Es gibt seit 2003 Algorithmen, die das Suffixarray direkt in linearer Zeit bauen
  - Wir besprechen gleich einen (einfacheren) direkten O(log(m)*m) Algorithmus

Suche mit Suffixarrays

- Finde alle Vorkommen eines Pattern P im Suffixarray für String T
- Ideen?
- Suche alle Vorkommen von P=„ssi“ in „mississippi“
  - Erinnerung: Jeder Substring ist Präfix (mindestens) eines Suffix
  - P liegt also am Anfang eines Suffix (wenn P in S)
  - Suffixe liegen alle sortiert vor
  - ➢ Binäre Suche im Suffixarray

```
a[1]=11    i
a[2]=8     ippi
a[3]=5     issippi
a[4]=2     ississippi
a[5]=1     mississippi
a[6]=10    pi
a[7]=9     ppi
a[8]=7     sippi
a[9]=4     sissippi
a[10]=6    ssippi
a[11]=3    ssissippi
```

Suchalgorithmus

```
l:=1; r:=m; n=|P|;
deffunc f(i):=S[a[i]..a[i]+n-1];
while l<r do
    z = floor(l+(r-l+1)/2);           // Middle of interval
    if (f(z) > P) then r = z;         // Go up
    else if f(z) < P then l = z;      // Go down
    else                              // Found one occurrence
        z‘:=z;
        while (f(z‘)=P & z‘>0) do     // Search “smaller” occ’s
            report a[z‘];
            z‘ := z‘-1;
        end while;
        z‘:=z+1;
        while (f(z‘)=P & z‘<=m) do    // Search “larger” occ’s
            report a[z‘];
            z‘ := z‘+1;
        end while;
        break;
    end if;
end while;
```

Beispiel

- Suche alle Vorkommen von P=„ssi“
- Komplexität?

```
a[1]=11     i
a[2]=8      ippi
a[3]=5      issippi
a[4]=2      ississippi
a[5]=1      mississippi
a[6]=10     pi
a[7]=9      ppi
a[8]=7      sippi
a[9]=4      sissippi
a[10]=6     ssippi
a[11]=3     ssissippi
```

Komplexität

- Wir machen im Worst-Case log(m) Versuche
- Jeder Test kostet im Worst-Case n Zeichenvergleiche
- Zusammen: O(n*log(m))
  - Sehr pessimistisch – jeder Vergleich müsste n-1 Matches haben, dann 1 Mismatch
- Wir unterschlagen gerade das Finden aller Vorkommen
  - Eigentlich: O(n*log(m) + k)
  - k: Anzahl Vorkommen von P in T

Beschleunigung

- Man muss nicht immer 1..n Zeichen vergleichen
- Fortlaufend zwei Positionen merken
  - p_l ist die Länge des längsten gemeinsamen Präfix von P und a[l] (linke Grenze des Suchintervalls)
  - p_r ist die Länge des längsten gemeinsamen Präfix von P und a[r] (rechte Grenze des Suchintervalls)
- Beim Vergleich mit dem mittleren Element des Intervalls
  - Sei $p=\min(p_r, p_l)$
  - Der String [1..p] ist gemeinsames Präfix aller Suffixen im Intervall
  - Vergleiche Zeichen erst ab Position p+1
- Gewinn
  - Keine veränderte Worst-Case Komplexität
  - Aber sehr gute Ergebnisse (quasi-linear) im Average Case
  - Algorithmen mit Worst-Case $O(n + \log(m))$ bekannt
    - Gusfield, p. 152-154

Algorithmus von Manber und Myers

Konstruktion von Suffix Arrays

- Bisher
  - In O(m) aus einem Suffixbaum – linear, aber Platzproblem bleibt
  - Man könnte auch alle Suffixe direkt aufzählen und sortieren
    - Braucht O(m*log(m)) String-Vergleiche, die jeweils bis zu m Zeichen vergleichen – schlecht
  - Aber die Suffixe enthalten sich doch alle irgendwie selber
- Das kann man ausnutzen
  - Manber and Myers. „Suffix arrays: a new method for online string searches". SIAM Journal of Computing, 22(5):935-- 948, 1993.
  - Komplexität O(m*log(m))
    - Im average case sogar O(m*log(log(m)))
- In 2003 auch erste Veröffentlichungen direkter, linearer Konstruktionsalgorithmen für Suffixarrays

Notwendig: Bucket-Sort

- Gegeben eine Menge von Strings
- Aufgabe: Sortiere diese nach dem ersten Zeichen
- Bucket-Sort sortiert Strings in Liste von Buckets
  - In jedem Bucket beginnen alle Strings mit demselben Zeichen
  - Die Buckets sind nach dem ersten Zeichen sortiert
  - Sehr ähnlich zu Hashen mit sortierter Hashtabelle
- Das ist $O(m)$ für m Strings
  - Wir ignorieren Platzprobleme
  - Einfach Liste mit $|\Sigma|$ leeren Buckets erzeugen, Strings einsortieren, leere Buckets löschen

Manber/Myers – Grundaufbau 1

- Wir erzeugen zuerst ein Array A mit den Suffixen der Länge nach absteigend sortiert
  - Genauer: Startpositionen der Suffixe
- A wird mit Bucket-Sort nach dem ersten Zeichen sortiert
- Dann sind die Buckets schon richtig sortiert – wir müssen aber noch innerhalb jedes Buckets sortieren
- Um nach den zweiten Zeichen in einem Bucket B zu sortieren
  - Jedes zweite Zeichen ist natürlich das erste Zeichen des nächstkleineren Suffix
  - Nach denen haben wir schon sortiert
  - Ausnutzen – erste Sortierung nachschlagen statt neu vergleichen
  - Das kann man für alle Buckets mit einem Durchlauf von A machen

Grundaufbau 2

- Damit haben wir neue (kleinere) Buckets, die alle nach den ersten 2 Zeichen sortiert sind
- Im nächsten Schritt sortieren wir nach dem 3. und 4. Zeichen unter Benutzung der vorhandenen Sortierung
- Dann nach dem 5-8 …. Etc.
- Man kann aufhören, wenn alle Buckets nur noch ein Suffix enthalten
- Das kann maximal log(m) Durchläufe benötigen
- Damit haben wir den Worst-Case: O(m*log(m))
  - Aber den Algorithmus noch nicht erklärt

Details

- Wir bauen das Suffixarray für S+"$"
- Array mit allen Suffixen von S der Länge nach absteigend initialisieren: A[i] = m +1 – i, i<m
- Mit Bucket-Sort nach dem ersten Zeichen sortieren
- Sortierung nach dem nächsten Zeichen erfolgt wie folgt:
  - Seien S[i..] = s_i … $s_m$$ und S[j..] = s_j … $s_m$$ zwei Strings in einem Bucket
  - Man müsste nun s_{i+1} mit s_{j+1} vergleichen
  - Das muss dasselbe Ergebnis ergeben wie der Vergleich von S[i+1..] = s_{i+1} … $s_m$$ mit S[j+1..] = s_{j+1} … $s_m$$
  - Diese Vergleiche haben wir schon alle gemacht und müssen nur noch das richtige Ergebnis im Array „finden"

Beispiel

```
123456789012
MISSISSIPPI$
```

$	12: $
I	2: ISSISSIPPI$ 5: ISSIPPI$ 8: IPPI$ 11: I$
M	1: MISSISSIPPI$
P	9: PPI$ 10: PI$
S	3: SSISSIPPI$ 4: SISSIPPI$ 6: SSIPPI$ 7: SIPPI$

- Sortierung der zweiten Zeichen entspricht Sortierung der ersten Zeichen an anderer Stelle im Array
- Setze in jedem Bucket einen Pointer auf das erste Element
- Wir gehen einmal durch das Array (i)
  - Sei A[i] = j. Dann ist S[j] das zweite Zeichen von dem Suffix an Position j-1
  - Da wir die i sortiert durchlaufen, ist das Suffix ab Position j-1 das nächst kleinere in seinem Bucket
  - Also schieben wir den Wert an den Startpointer seines Buckets und verschieben den Startpointer um eins

Beispiel 2

- In jedem Bucket wird nach dem zweiten Zeichen sortiert
- Durch das Array laufen. Sei A[i]=j. Dann schiebe den Index j-1 an den Anfang in seinem Bucket
- Alle Buckets werden dabei gleichzeitig nach dem zweiten Zeichen sortiert
- Oftmals entstehen neue Buckets

A	1 12	2 3 4 5 2 5 8 11	6 1	7 8 9 10	9 10 11 12 3 4 6 7
12		11			
2			1		
5					4
8					4 7
11				10	
1					
9		11 8			
10				10 9	
3		11 8 2			
4					4 7 3
6		11 8 2 5			
7					4 7 3 6
	12	11 8 2 5	1	10 9	4 7 3 6
	$	I I I I $ P S S P S S	M I S	P P I P $ I	S S S S I I S S S P I I

Sortierung nach 3. und 4. Zeichen

- Bisher
  - Alle Elemente eines Buckets haben das gleiche, 2-buchstabige Präfix
  - Alle Buckets liegen sortiert nach dem Präfix im Array
- Nächste Schritte
  - Seien S[i..] = s_i … $s_m$$ und S[j..] = s_j … $s_m$$ wie vorher
  - Man müsste nun s_{i+2} mit s_{j+2} vergleichen
  - Das ist das gleiche wie S[i+2..] = s_{i+2} … $s_m$$ mit S[j+2..] = s_{j+2} … $s_m$$
  - Die S[i+2..] und S[j+2..] haben wir schon sortiert, und zwar nach den ersten beiden Zeichen
- Wir sortieren simultan nach dem 3. und 4. Zeichen
  - Wir gehen wieder durch das Array
  - Sei A[i] = j. Dann ist S[j] das dritte Zeichen von dem Suffix S[j-2,…]
  - Schiebe den Index j-2 an den Startpointer seines Buckets und verschieben den Startpointer um eins
  - Das sortiert automatisch nach dem 3. und 4. Zeichen
- Dann nach 5-8. Zeichen
- Etc.

Beispiel

- Ob neue Buckets entstehen, kann man zur Laufzeit mitmerken
- Aufhören, wenn alle Buckets nur noch ein Suffix enthalten

A	1	2	3	4	5	6	7	8	9	10	11	12
	12	11	8	2	5	1	10	9	4	7	3	6
12							10					
11								9				
8											6	
2												
5											6	3
1												
10			8									
9									7			
4				2								
7				2	5							
3						1						
6									7	4		
	12	11	8	2	5	1	10	9	7	4	6	3
	$	I	I	I	I	M	P	P	S	S	S	S
		$	P	S	S	I	I	P	I	I	S	S
			P	S	S	S	$	I	P	S	I	I
			I	I	I	S		$	P	S	P	S
			$	S	P	I			I	I	P	S

Zusammenfassung

- Suffixarrays
  - Sehr wenig Speicher (m Bytes)
  - Sehr schnelle Suche (O(n+log(m))
  - Direkte Konstruktion in O(m*log(m)) (oder besser)
- Aber
  - Viele schöne Tricks für Suffixbäume funktionieren nicht
  - Warum? Weil man die „Least common ancestors“ (längste gemeinsame Präfixe = tiefster gemeinsamer Vaterknoten) nicht kennt
  - Zum Glück kann man die in linearer Zeit berechnen
- Enhanced Suffixarrays: Suffix Array + LCP Tabelle
  - Äquivalent zu Suffixbäumen für die meisten Anwendungen

Aber ...

- Exakte Suchalgorithmen sind ja gut und schön ...
  - Z-Box, Boyer-Moore, Knuth-Morris-Prath, Aho-Corasick, Suffixbäume, Suffixarrays
- ... aber häufiger braucht man unscharfe Vergleiche
  - Mismatches, Deletions, Insertions
- Das kommt im nächsten Block