接尾辞木について

渋谷

東京大学医科学研究所ヒトゲノム解析センター
（兼）情報理工学系研究科コンピュータ科学専攻
http://www.hgc.jp/~tshibuya

文字列索引

- 文字列探索を効率化するためのデータ構造のこと
  - テキスト文字列に対して前処理を行うことによって得る
  - 厳密マッチング
    - 接尾辞木、接尾辞配列、FM-index
  - 非厳密マッチング
    - FASTA、BLAST （いずれもヒューリスティック）

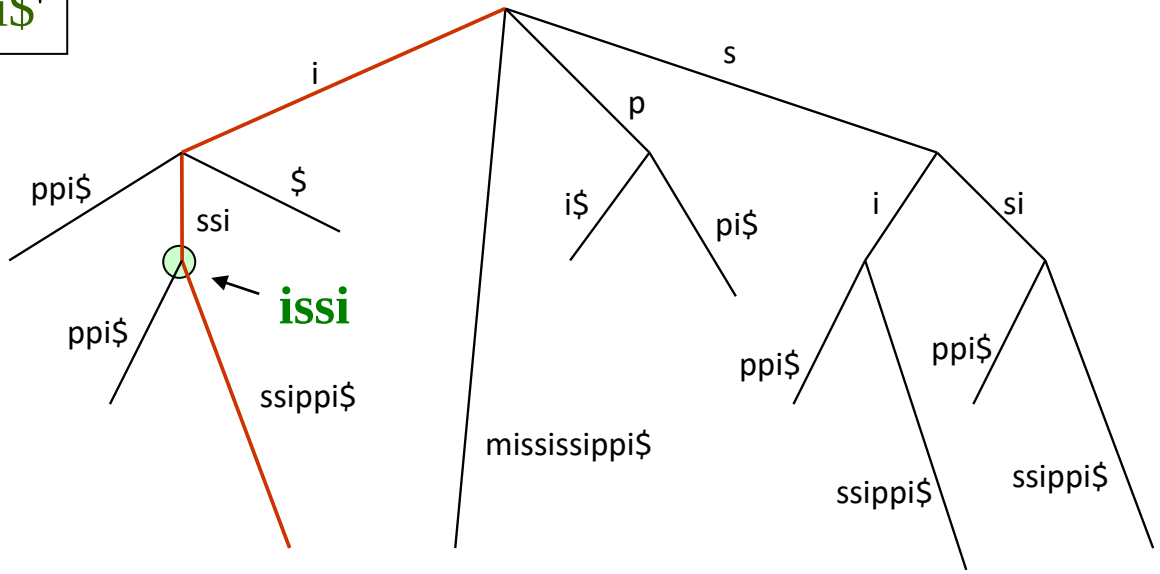
接尾辞木とは

- 文字列 S のすべての接尾辞を表した trie
  - 枝のラベル ⇔ S の部分文字列
  - ルートから葉までのラベルを連結したもの ⇔ S の接尾辞

Suffix tree of 'mississippi$'

All the suffixes

mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$



$について

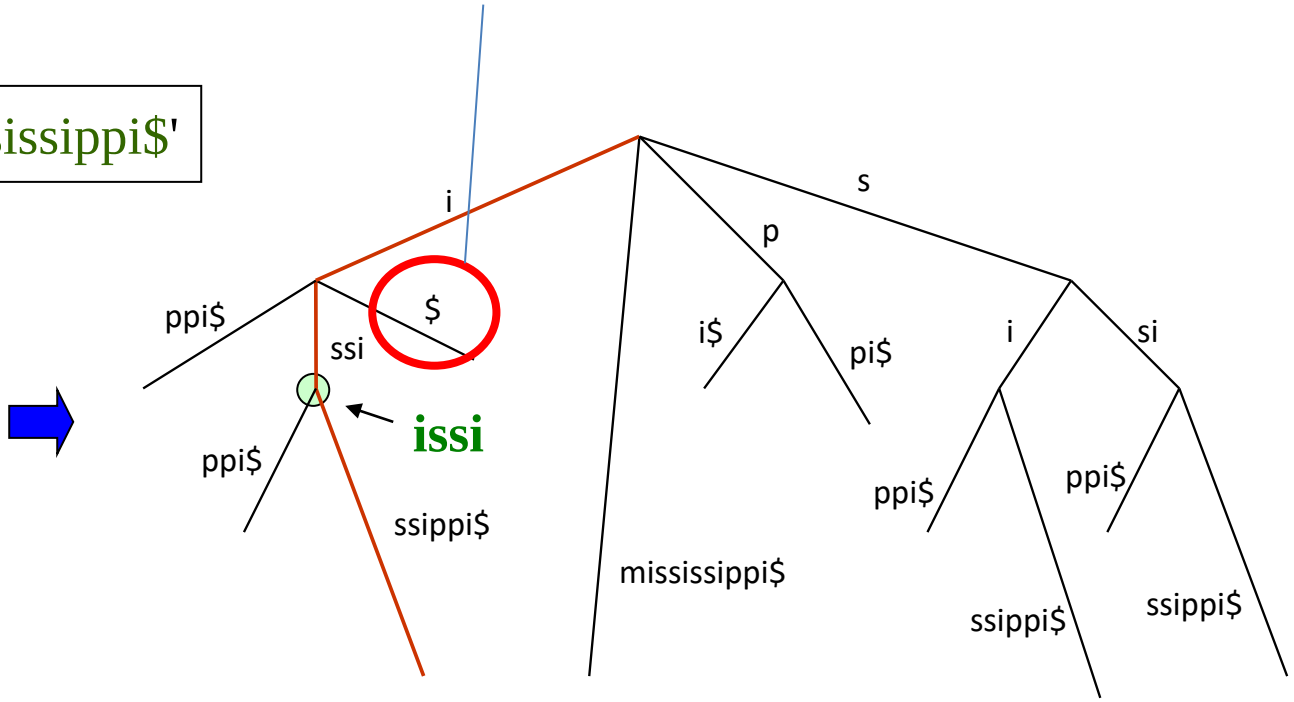
- テキスト・パタンに含まれない文字
  - 接尾辞木に限らず、様々な場面で有効に用いられる
  - 多くの場合、終端記号として用いられる

$があれば、すべての接尾辞と葉ノードが1:1に対応する

Suffix tree of 'mississippi$'

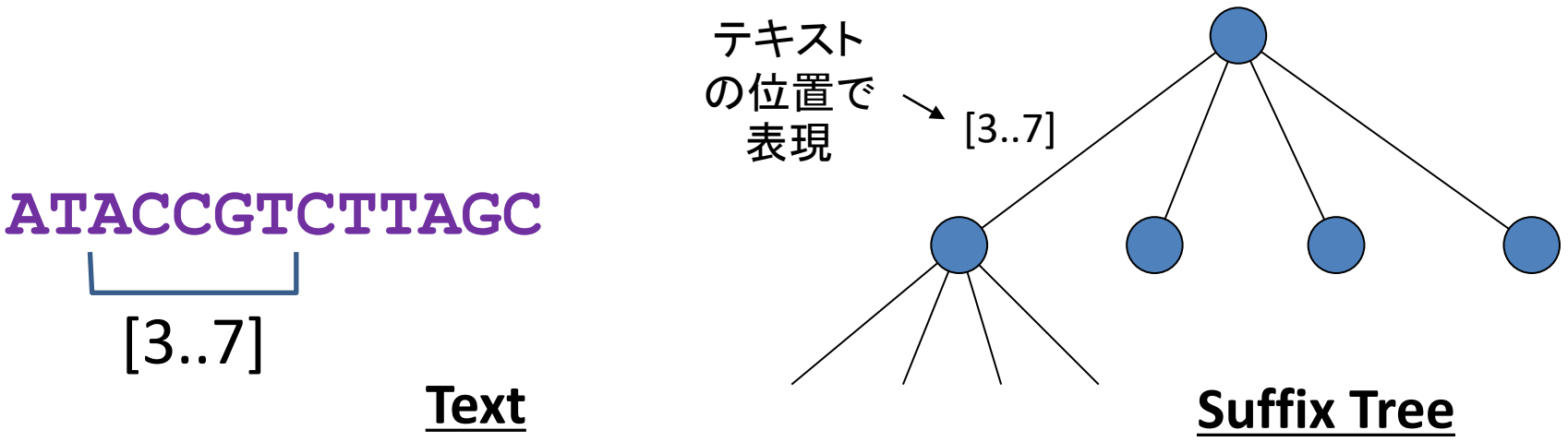
All the suffixes

mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$



接尾辞木のサイズ

- もし、ナイーブにラベルを格納したとすると？
  - $O(n^2)$のサイズが必要になる
- うまくやれば全体で$O(n)$のサイズで表現できる
  - 枝のラベルは文字列のインデックスで表現可能
    - 1つのノードあたりコンスタントメモリー（たった2つの整数）
    - テキスト自体は別に覚えておく
  - 葉の数（＝接尾辞の数＝文字列の長さ）は n
  - 全体のノードの数（＝枝数＋1）は $2n-1$ 以下
    - 内部ノードの次数は2以上なので

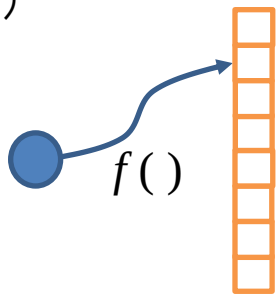


ノードの表現について

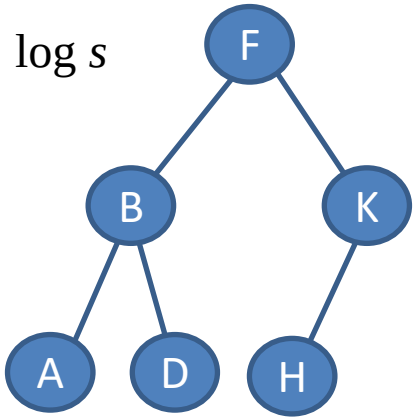
- 様々なノードとその子ノードの表現
  - リスト　（もっとも一般的）

    - 各ノードが子供へのポインタと次の兄弟へのポインタを持つ
    - ソートする場合としない場合がある
    - 最悪の場合、子供を辿る際に子供をすべてチェックする必要がある
  - 配列

    - アルファベットサイズが小さい場合にはOK
    - ソートした配列を持てば二分探索が可能（ただし更新は非効率的）
  - ハッシュ

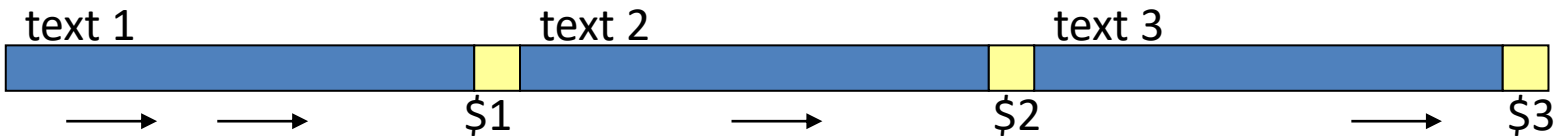


    - 文字数が多い場合によくする実装
    - 木をすべてtraverseするのに不利
      - となりのノードを見つけるのに余計な時間がかかる（ことが多い）
  - 二分木（様々なバリエーションが存在する）
    - ラベルを辿る際にアルファベットサイズ s に関する計算量を $\log s$ のオーダーに抑えることができる
      - 子供を追加するといった、更新も同じ計算量で可能
      - ラベルと関係なく子供を辿る場合は、$O(1)$
    - 実際の実装ではなかなか使われない
      - 実装に必要なメモリ量が大きすぎる



接尾辞木の特技

- 構築：なんと線形時間！！
- 検索：もともとあるテキストに対して、あるパタンがあるかどうか、あればどこにあるかを探す
  - パタンの長さに対して線形時間　（アルファベットサイズ：固定）
  - 木を上から辿る
- 頻出部分文字列検索：あるテキスト中で頻出する部分文字列を探す
  - 線形時間
  - 単にノードに対応する文字列を列挙するだけ
- 共通部分文字列探索：複数のテキスト中で共通に出現する部分文字列を探す
  - これも線形時間
  - 異なる終端文字を加えてから連結し、接尾辞木を作成する



他にもたくさんの応用がある。
Gusfieldの本に詳しい

接尾辞木の作成アルゴリズムの歴史

歴史

- Weiner ’73
  - $O(n\,s)$（s: アルファベットサイズ）
- McCreight ’76
  - $O(n \log s)$
- Ukkonen ’95
  - $O(n \log s)$
  - McCreightの簡単化・On-line化
- Farach ’97
  - $O(n)$ for integer alphabet $[1..n]$
- 接尾辞配列から作成
  - $O(n)$ for integer alphabet $[1..n]$
  - Kasai et al. ’00 + Kärkkäinen & Sanders ’03, etc.
    - 逆に接尾辞木から接尾辞配列は、自明に線形時間で作成可能。

[Gusfield '97]に詳しい

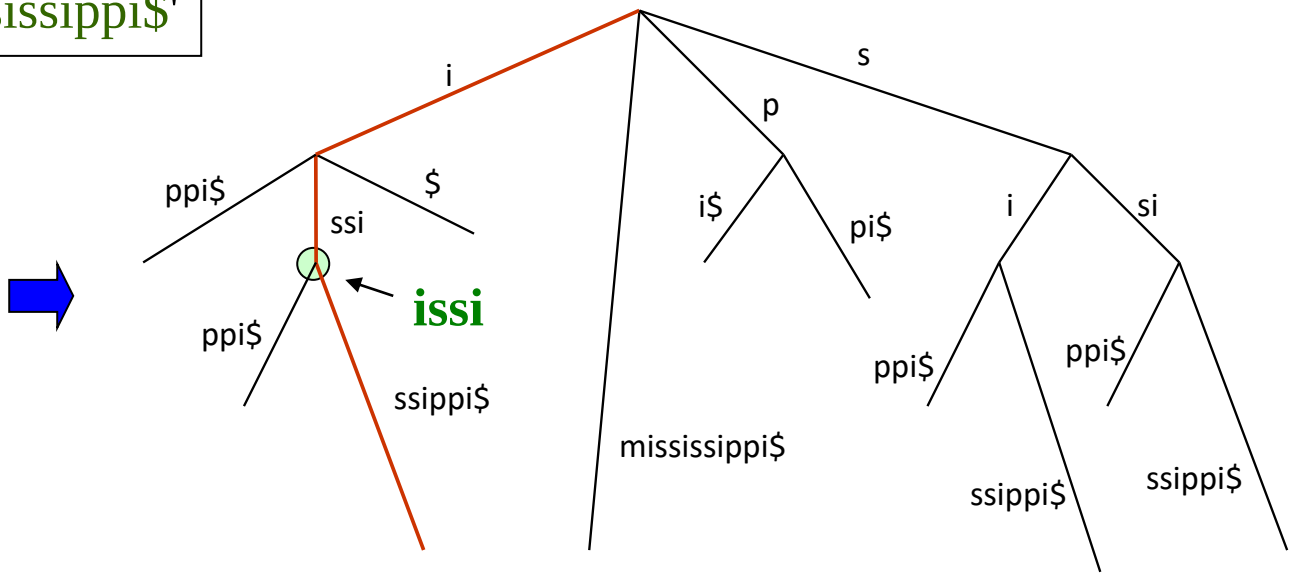
ナイーブに作ると？

- ひとつの葉を挿入するのに$O(n \log s)$時間
  - もっとナイーブに作ると$O(n \cdot s)$時間
  - n: 文字列の長さ　s: アルファベットサイズ
- 全体で$O(n^2 \log s)$　時間　→　遅すぎる！

Suffix tree of 'mississippi$'

All the suffixes

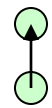
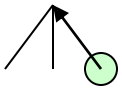
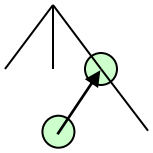
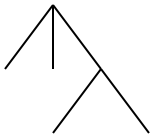
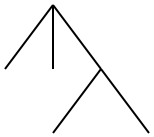
mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$



Ukkonen's Algorithm (1) [Ukkonen '95]

- $S[1..n]$の接尾辞木の作成方法
  - 左から順番に作っていく
  - Phase i
    - $S[1..i]$にノードを(必要なら)挿入することで$S[1..i]$の接尾辞木を作る
  - $O(n \log |\Sigma|)$ time (n : string length, Σ : alphabet)

Ukkonen Algorithm for 'mississippi$'

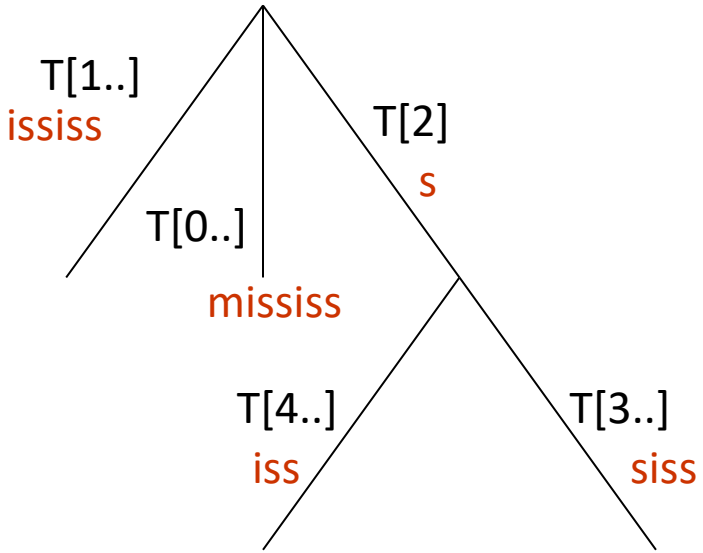
Phase	1	2	3	4	5	6	....
Suffixes	m	mi i	mis is s	miss iss ss s	missi issi ssi si i	missis issis ssis sis is s	●●●
Ti							●●●

Ukkonen's Algorithm (2)

- Implicit Suffix Tree
  - 葉に関してはテキスト上の始めの位置のみを記憶
    - 右に伸ばした時に、変更する必要をなくするため。

mississippiの接尾辞木を作成中

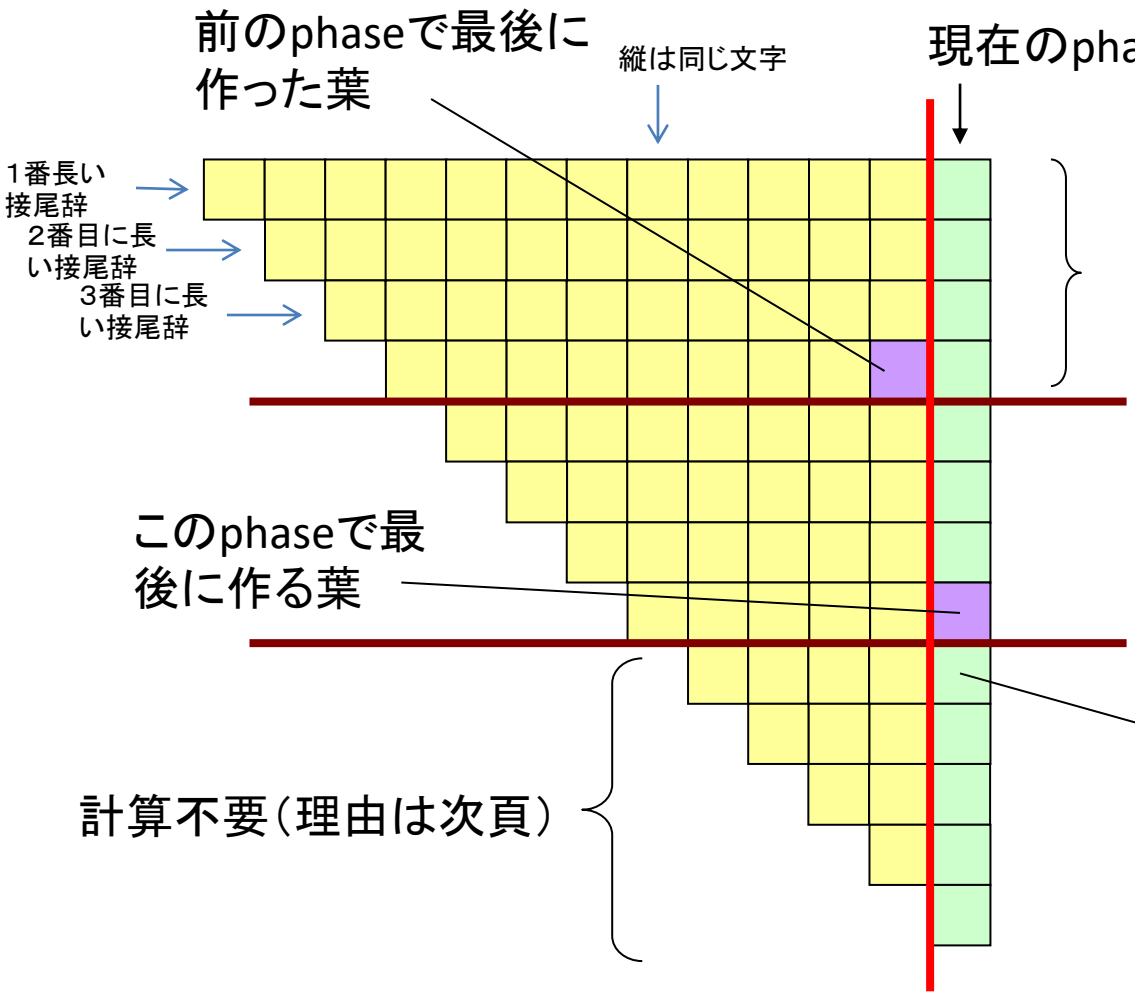
mississ
missis
issis
ssis
sis
is
s



次はiを加えることになるが、
ノードを追加する必要はない

Ukkonen's Algorithm (3)

各phase で葉を追加する必要がある接尾辞は？



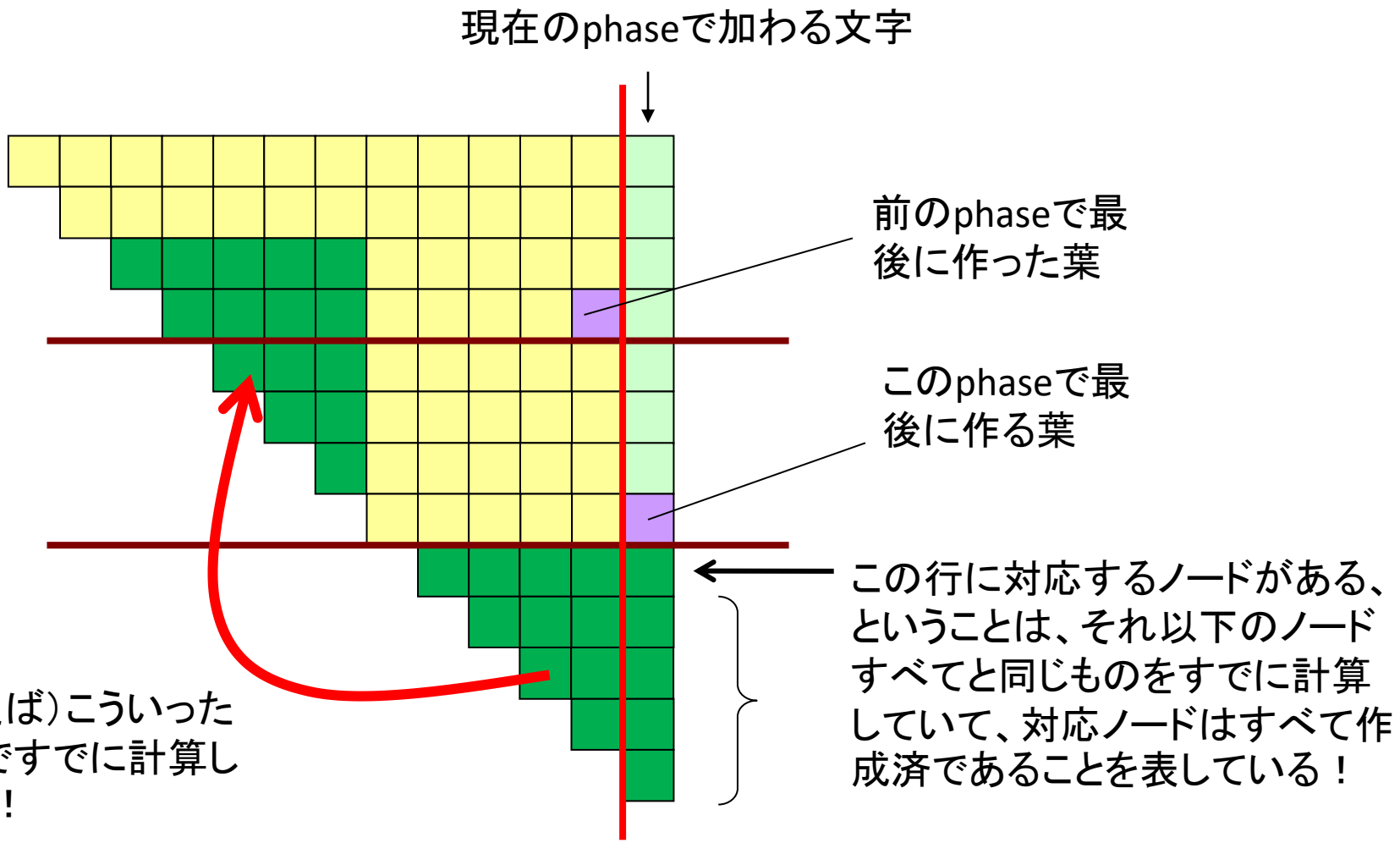
すでに対応する葉ノードを作成済。
implicit suffix treeなので、ラベル
の変更も必要もない。

相当する場所が存在するか
チェックして、存在しない場
合は、対応する葉ノードを新
規に作成する

この文字列に相当する場所
がすでに木に存在していたな
らば、ここより下はこのphase
では計算しないでよい。
次のphaseはこの行から下で
計算を開始する。

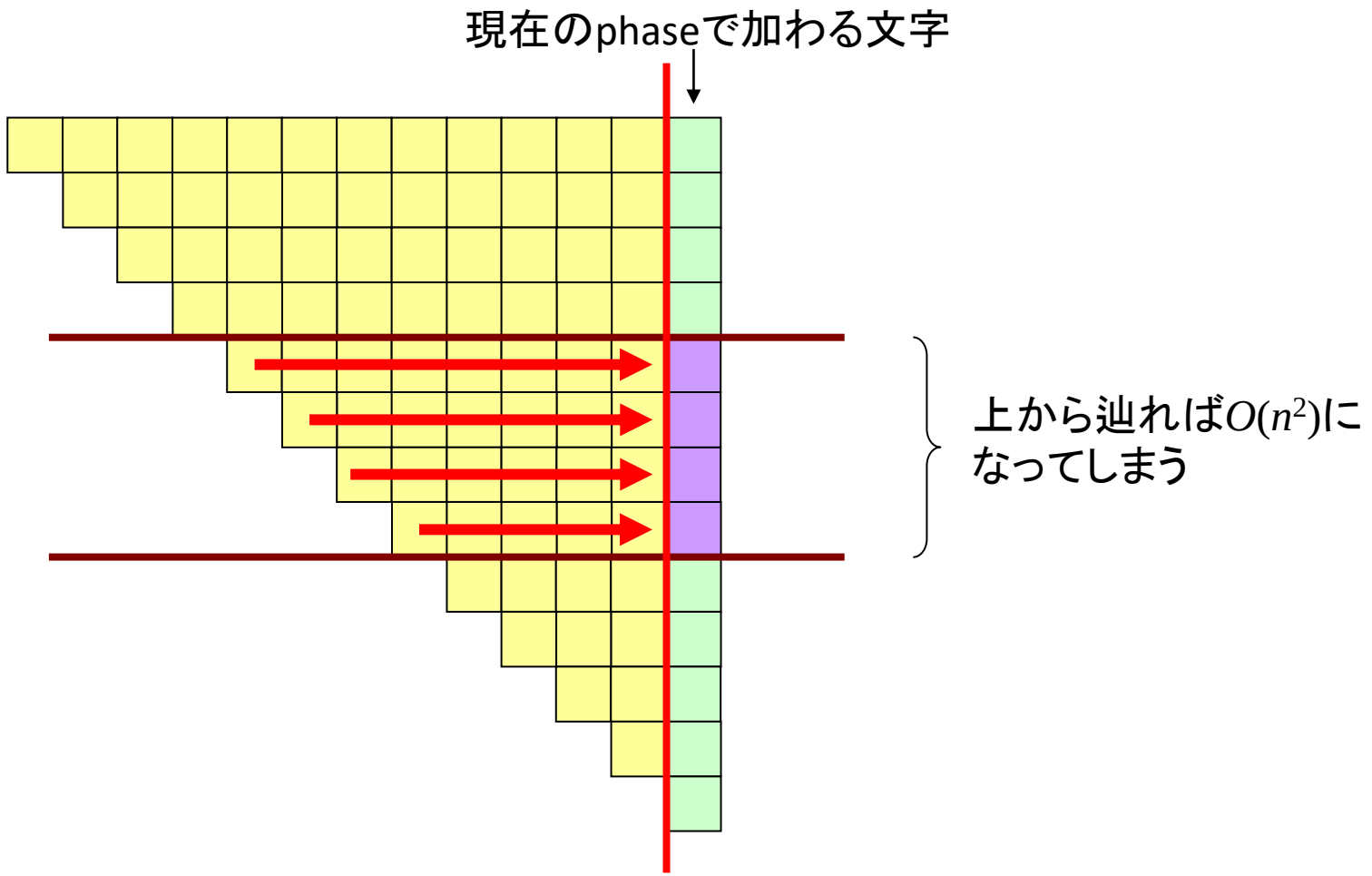
Ukkonen's Algorithm (4)

- なぜ、下は計算しなくてよいのか？



Ukkonen's Algorithm (5)

- しかしながら、ノードを加える時に木をルートから辿って作成すると、線形時間にはならない！→ suffix links



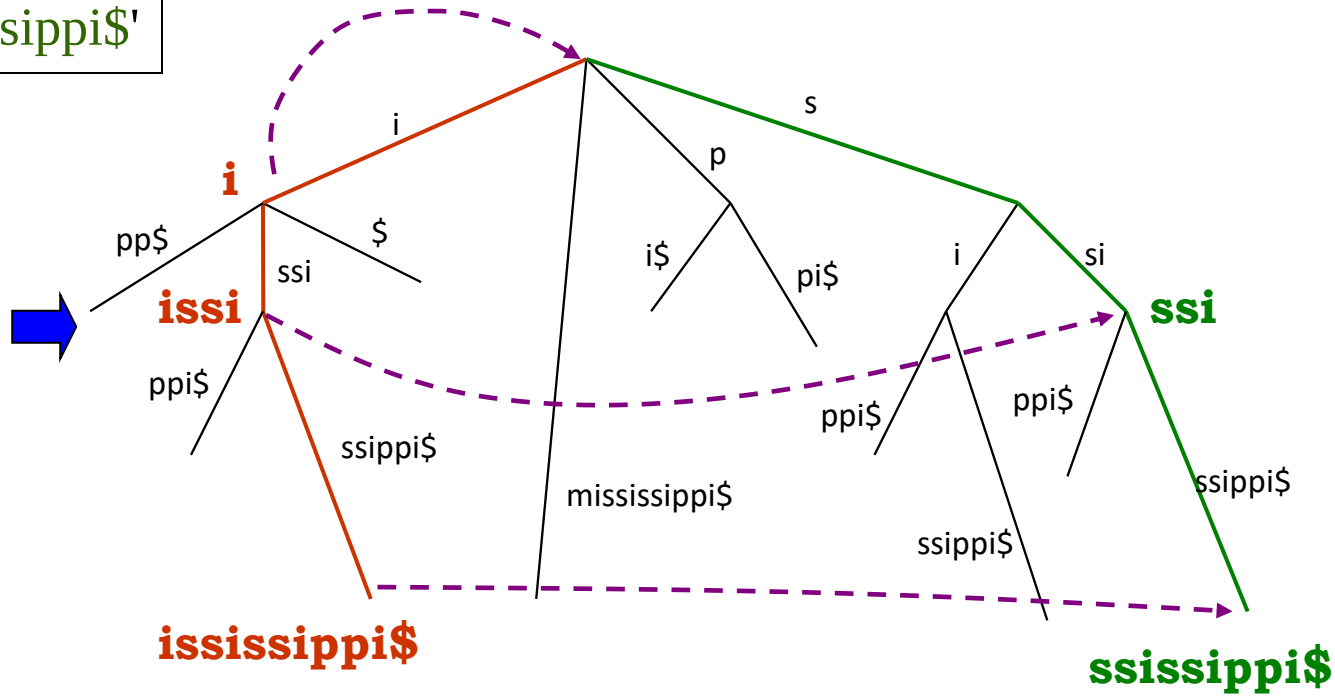
Ukkonen's Algorithm (6)

- Suffix Links
  - ノード p から p より1だけ短い接尾辞 q へのポインタ
    - Lq is a suffix of Lp
    - $|Lq| = |Lp|-1$　　(Lp: label of p,　Lq: label of q)
    - 理論的に必ず存在する
  - 一つ短いsuffixの位置を知ることができる。
  - 他にも様々な応用がある(最大共通部分文字列の計算など)

Suffix tree of 'mississippi$'

All the suffixes

mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$



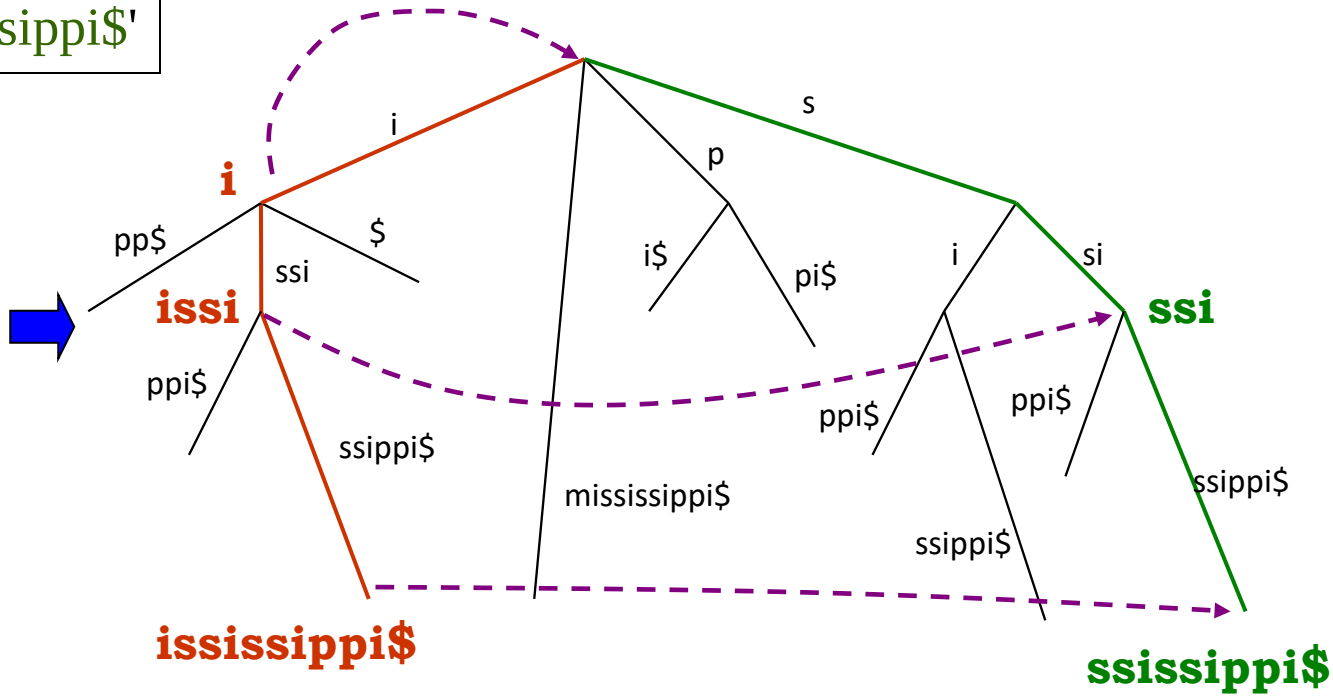
Ukkonen's Algorithm (7)

- なぜsuffix linkが「必ず」存在するのか？
  - あるノードが存在する　＝　そのノードのラベル（そこまでの枝のラベルの連結したもの）で始まり、その次の文字が異なる部分文字列が2組以上存在する、ということ
  - 当然、その接頭辞1文字を除いた部分文字列の組も存在するため、それに対応したノード、すなわちsuffix link先も存在する

Suffix tree of 'mississippi$'

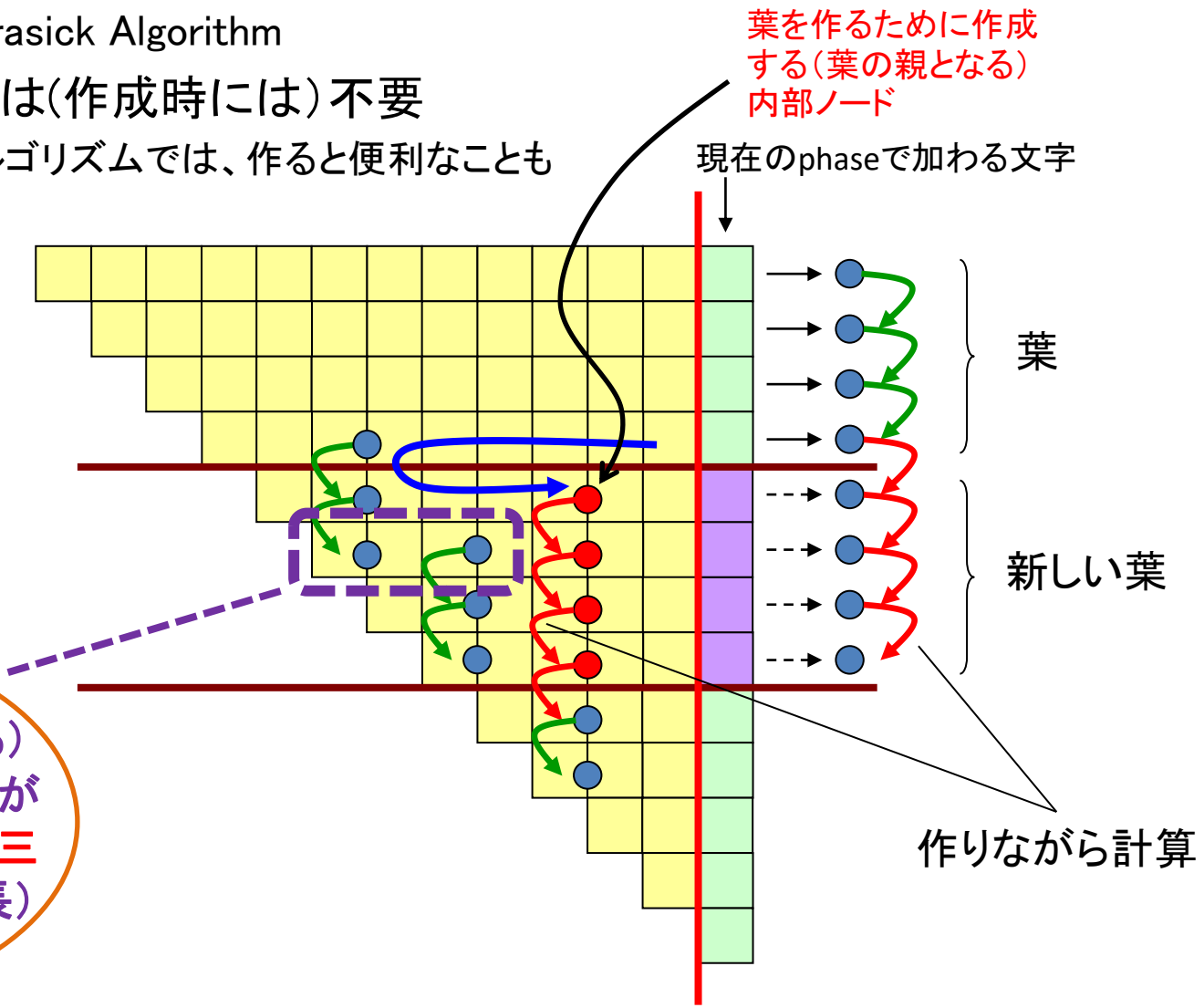
All the suffixes

mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$



Ukkonen's Algorithm (8)

- Suffix Linkの利用
  - Suffix linkを辿れば、全体の比較回数は$O(n)$ですむ
    - *cf.* Aho-Corasick Algorithm
  - 葉のsuffix linkは(作成時には)不要
    - 但し応用アルゴリズムでは、作ると便利なことも



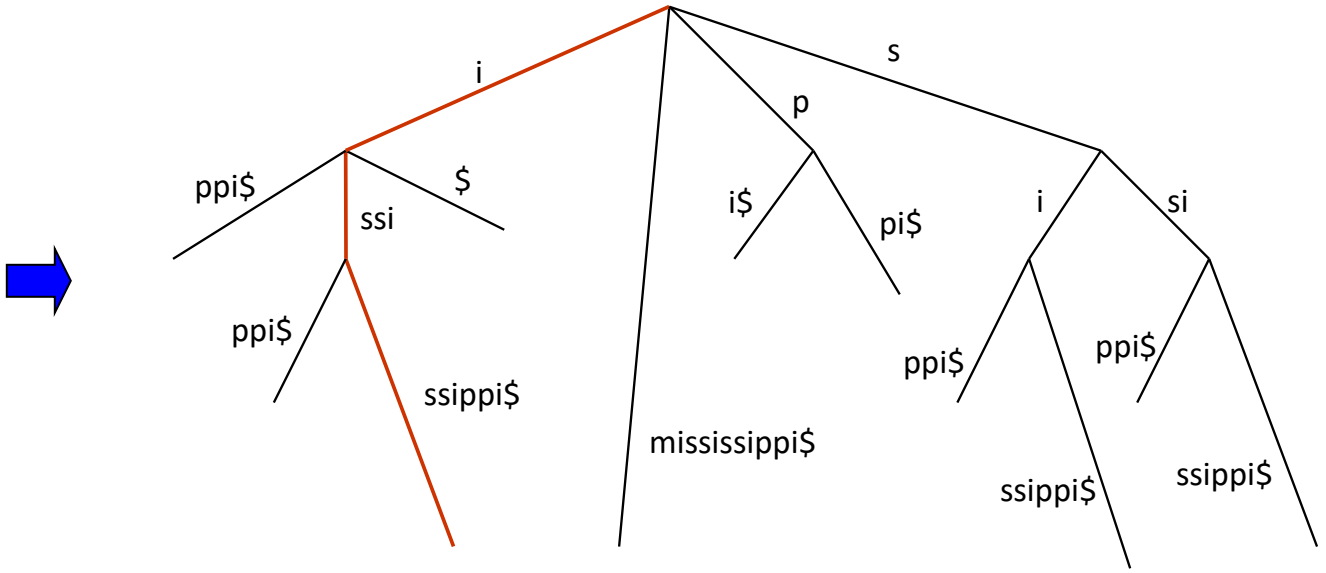
Ukkonen's Algorithm (9)

- 全体の計算時間
  - $O(n \log s)$
    - sはアルファベットサイズ
    - アルファベットを平衡木でアクセスするようにした場合
    - ハッシュで表せば平均的に$O(n)$
    - 配列で持つ場合は初期化するだけで$O(ns)$必要
    - 子供をリストで持つ場合も$O(ns)$

McCreight Algorithm（概略のみ）

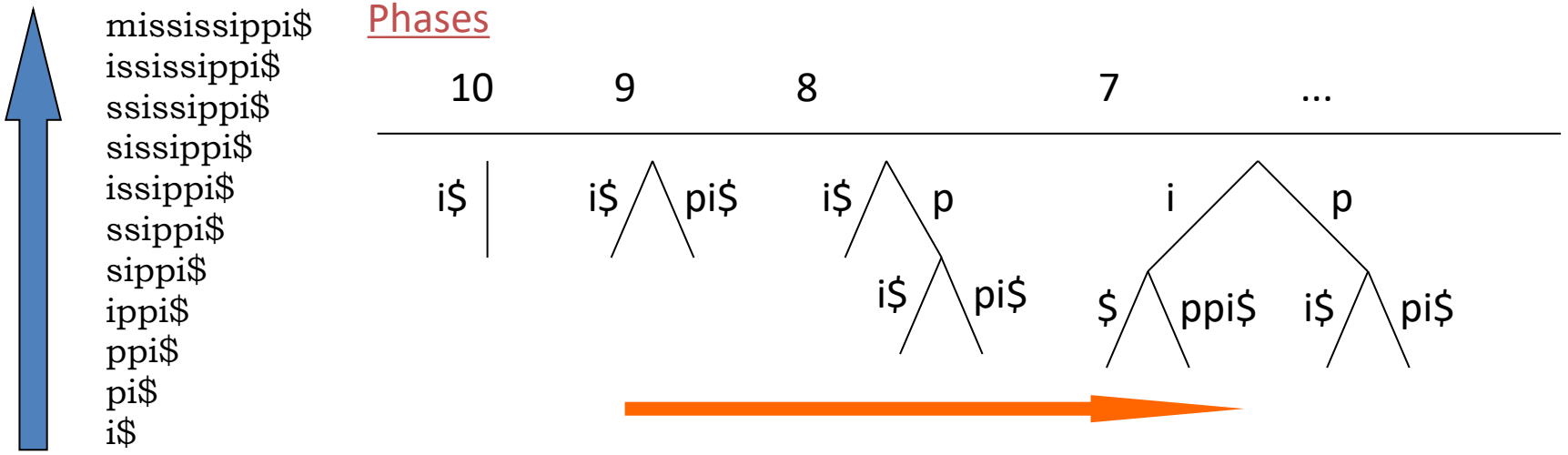
- アルゴリズムとしてはUkkonenと殆ど同じ
  - 一つ目のsuffix から順に挿入していくのは同じ
  - Ukkonenと違って、suffixは末尾まで考えていた
    - したがってMcCreightでは、テキストの文字が1字1字来る度に接尾辞木をupdateすることはできない
  - Suffix linkを活用することで計算時間はUkkonenと同じ
    - 活用方法も理論解析もUkkonenのアルゴリズムとほぼ同様

mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$



Weiner's Algorithm (1)

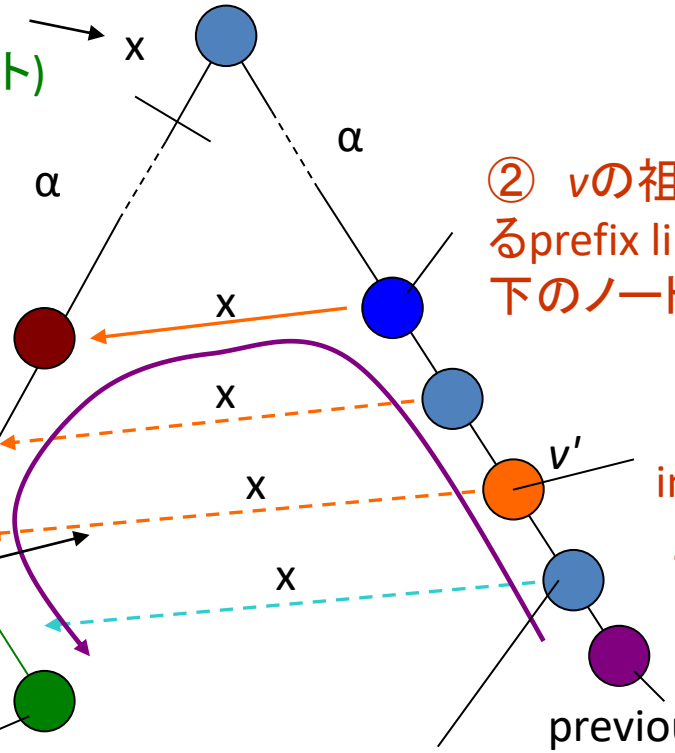
- Ukkonen と逆方向に接尾辞木を作成する
  - $O(n \cdot s)$のメモリー・計算時間が必要 (s: alphabet size)
  - ナイーブにすればやはり$O(n^2)$かかってしまう。



Weiner's Algorithm (2)

- suffix linkの逆のリンク(prefix link)を作成すればよい
  - 各ノードで最大アルファベットサイズの数のリンクが必要
  - 各ノードにフラグindicator[x] を用意
    - x はprefix linkに対応するアルファベット（すなわちアルファベットサイズ分必要）
    - 初期状態ではフラグは何も立っていない状態とする

新たに加える接尾辞の１文字目
（prefix linkに対応するアルファベット)



② vの祖先の中で、xに対応するprefix linkを持っている一番下のノードを見つける

③ この間にノードはない。一文字目だけをチェックして枝を選択。

① v の祖先の中でindicator[x]のフラグが立っている一番下のノードv'を探す（⑤で必要）

④ このノードv''を作成

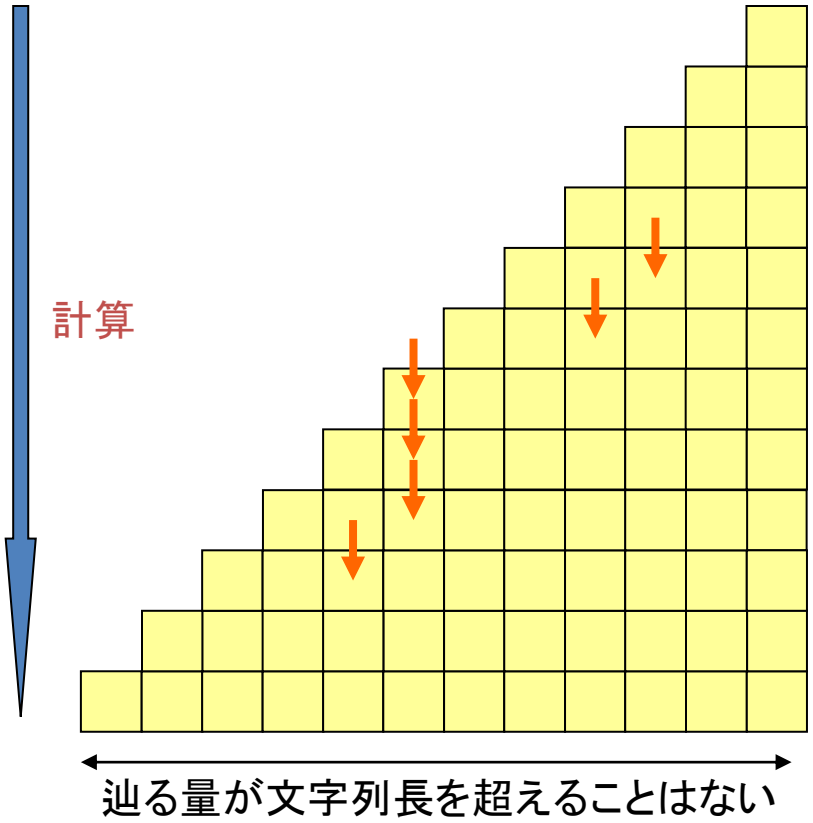
⑤ v'からv''へのprefix link を作成

⑥ 新しい葉を作成

⑦ vとv'の間のノードのindicator[x] フラグを立てる

Weiner's Algorithm (3)

- 計算時間
  - $O(n \cdot s)$ （sはアルファベットサイズ）
  - 解析はUkkonen と全く同じ



Farach's Suffix Tree Construction Algorithm

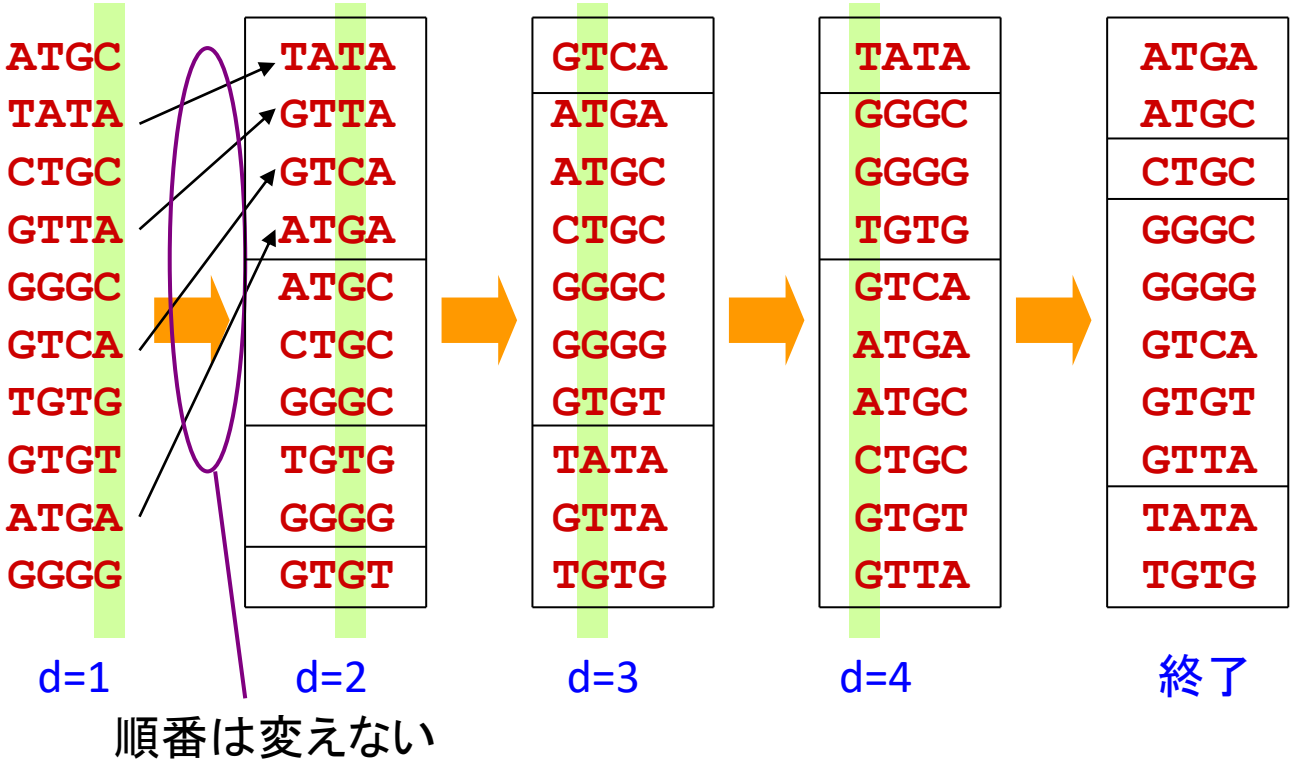
[Farach '97]

- Integer alphabet $\{1,\ldots,n\}$ に対して線形時間
  - 現実的にはあまり速いアルゴリズムとはいえない
    - インプリすること自体相当難しい
  - しかし、これが後のsuffix array構築アルゴリズムの革命的発展の端緒となった！
  - $\{1, \ldots, poly(n)\}$でも線形時間
    - radix sortで $\{1, \ldots, n\}$ の問題に持ち込めるため
- 3ステップからなる
  - Step 1
    - 半分のサイズの問題をとく（再帰的に計算）
  - Steps 2 and 3
    - その問題から線形時間で本物を構築
  - Computing time : $f(n) = f(n/2) + O(n) \rightarrow f(n) = O(n)$
    - 証明:

$$f(n) < f(n/2)+c\cdot n < f(n/4)+c\cdot n+c\cdot n/2 < f(const) + 2c\cdot n$$

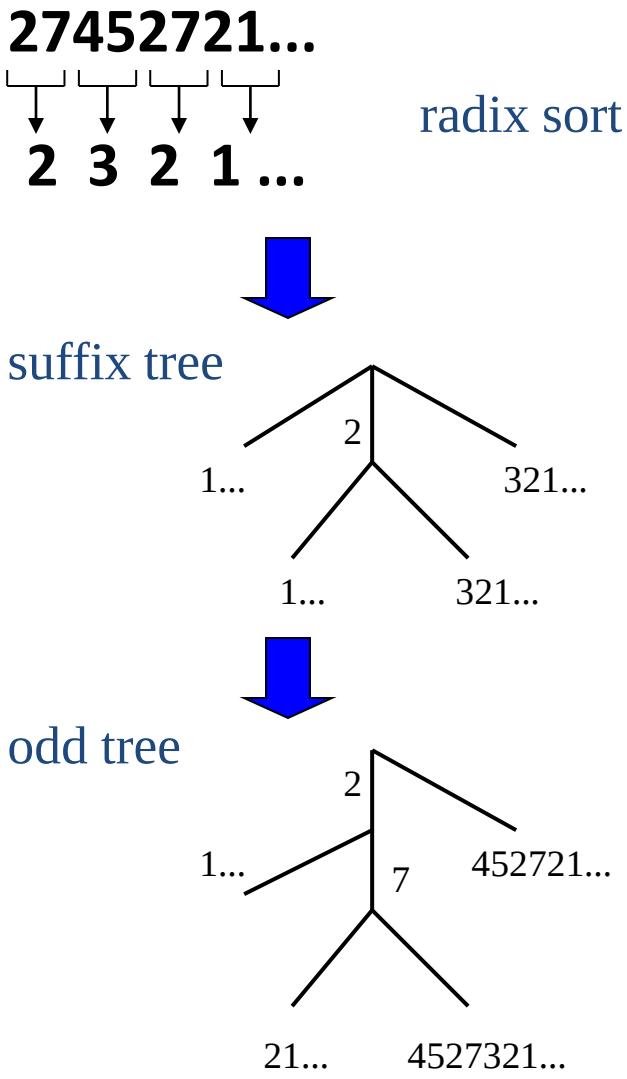
Radix Sort（復習）

- ラディックス（基数）・ソート
  - 下の位（あるいは文字）からソートして行く
    - その桁の文字が同じものについては順番を変えないようにする
    - バケツソートで $O(n+s)$　（sはアルファベットサイズ）
  - 計算量　$O((n+s)b)$　　（bは桁数）



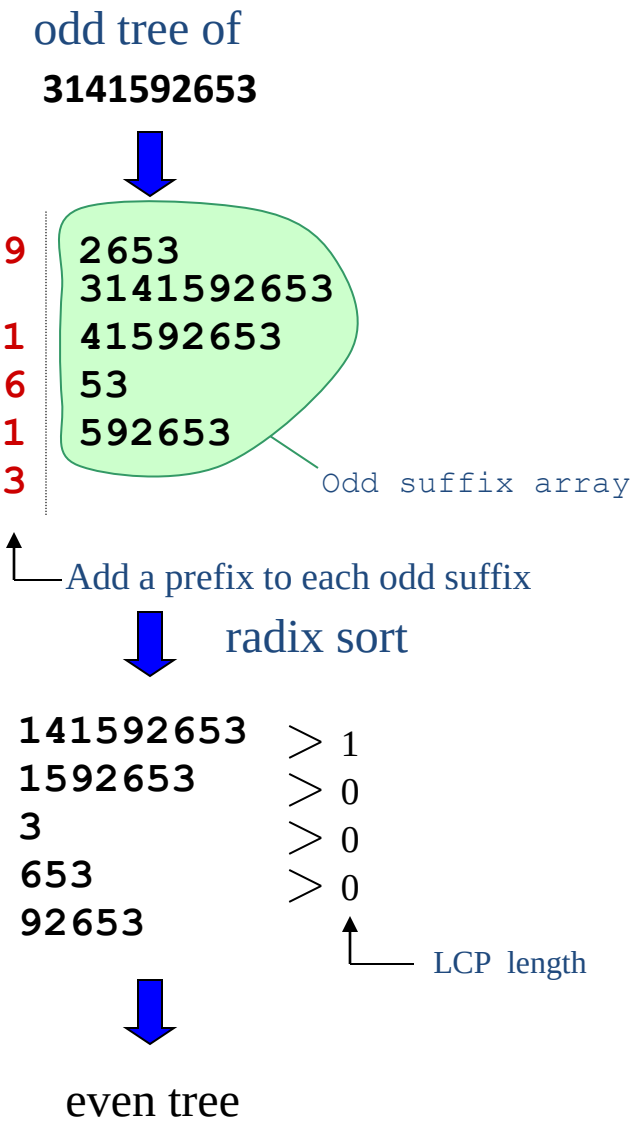
Step 1

- Odd Treeの作成
  - 奇数番目の接尾辞のtrie
  - 基数ソートを行い、半分の長さの文字列を作成
  - それに対して接尾辞木を作成
    - $f(n/2)$ time
      ($f(n)$: time for constructing suffix tree of a string of size n)
  - それを修正して奇数番目の接尾辞のtrieを作成（odd tree）
    - $O(n)$ time



Step 2

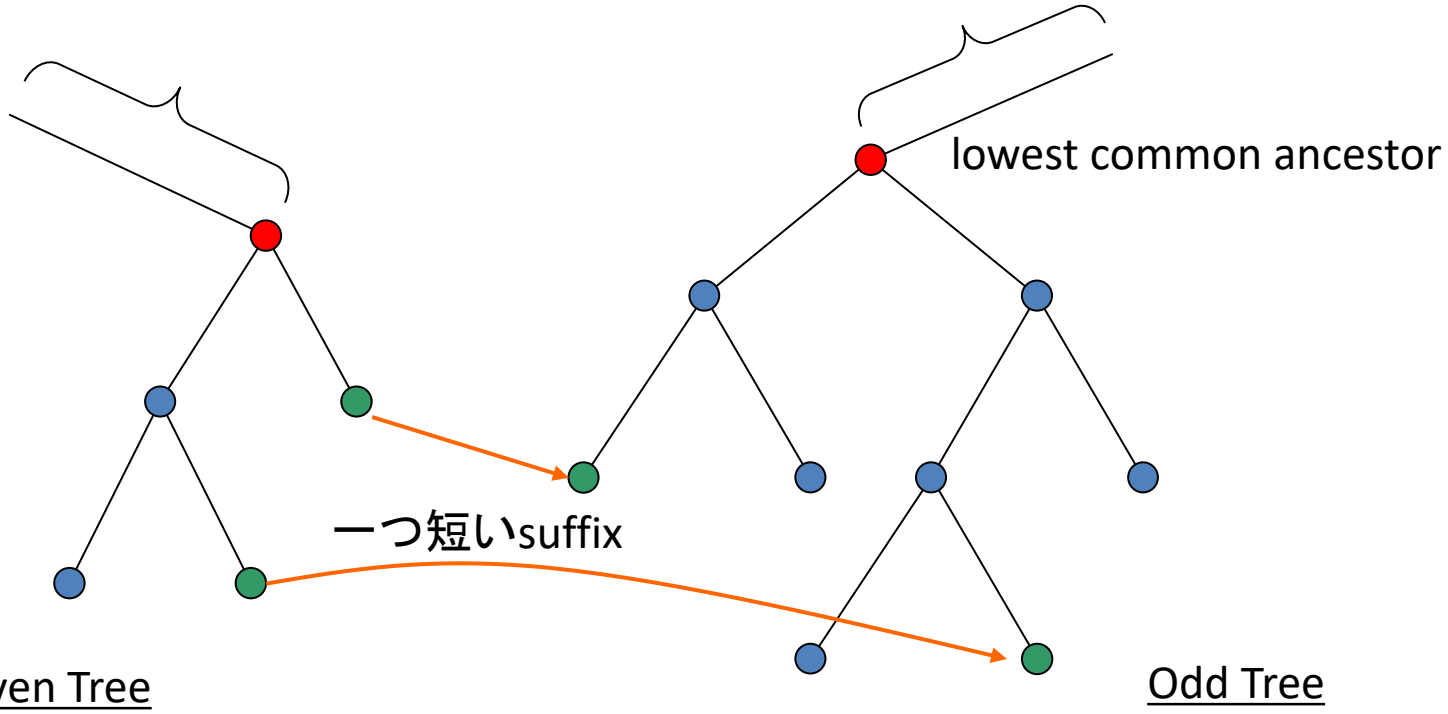
- Even Treeの作成
  - 偶数番目の接尾辞のtrie
  - 偶数番目のみの接尾辞配列を作成
    - 奇数番目の接尾辞配列から基数ソート
  - 偶数番目の接尾辞のtrieをそれから作成する
    - となり同士のLCP lengthを利用
      - LCP: Longest Common Prefix
      - これはOdd treeから計算する
  - $O(n)$ time



Step 2 (cont.)

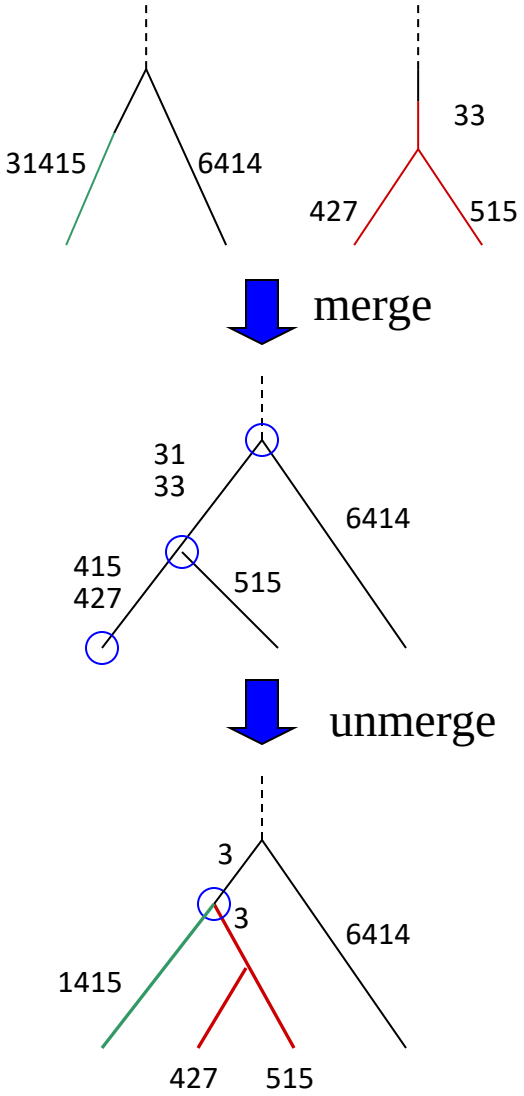
- となりのLCPを求めるには？
  - Odd tree 中の対応するLCAを求めることにより可能
    - これは$O(n)$の前処理で$O(1)$で可能！（後述）

$$|\mathrm{lcp}(l_{2i}, l_{2j})| = \begin{cases} |\mathrm{lcp}(l_{2i+1}, l_{2j+1})| + 1 & \text{if } s_{2i} = s_{2j} \\ 0 & \text{if } s_{2i} \neq s_{2j} \end{cases}$$



Step 3

- 2つの木をマージする
  - まずは枝の最初の文字だけ見てマージ
  - マージしすぎたものを戻す
    - マージした2つの文字列の本当のLCPを「うまく」計算して、マージした木と矛盾していれば、正しい状態に修正する
  - $O(n)$ time

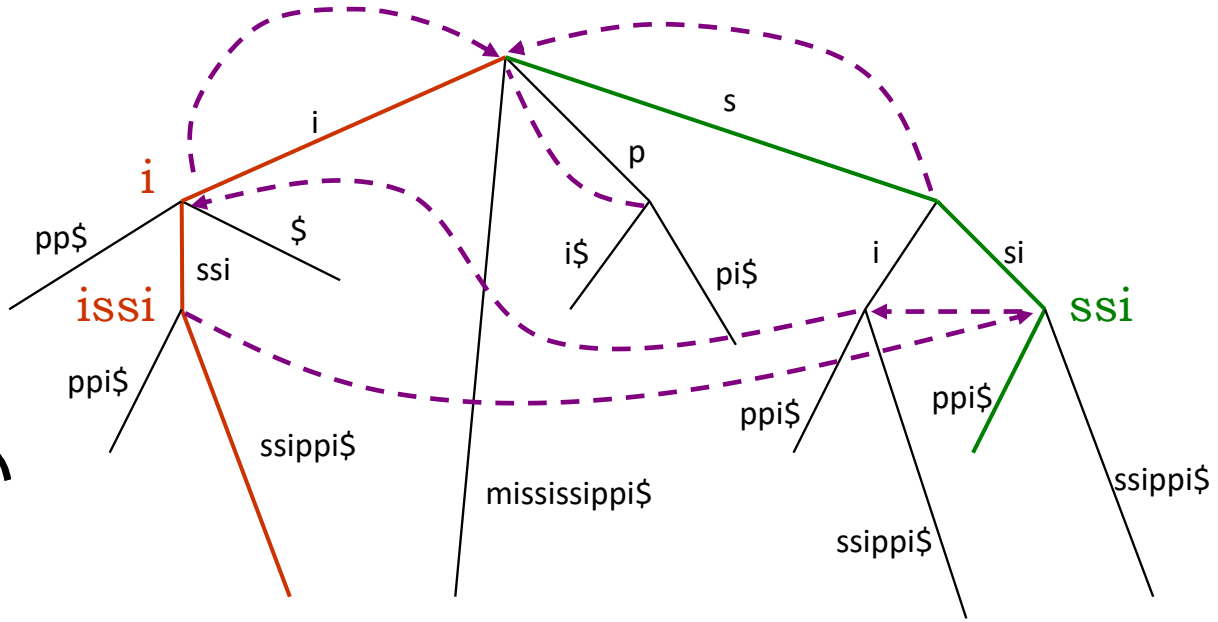


Step 3 (cont.)

- うまいlcpの計算法
  - 子供にodd/even両方の葉を含むノードについてodd/evenに相当する子供をそれぞれ1つずつ選ぶ

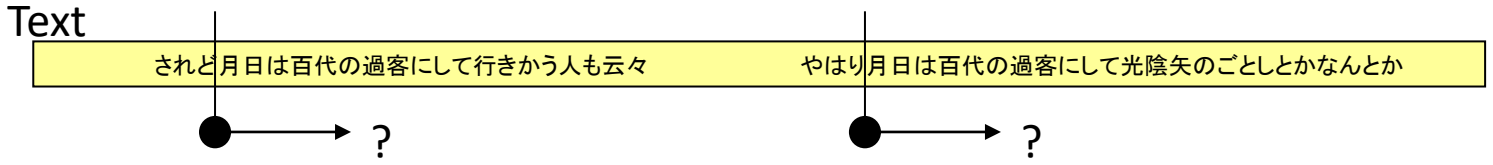
$$|\mathrm{lcp}(l_{2i}, l_{2j-1})| = \begin{cases} |\mathrm{lcp}(l_{2i+1}, l_{2j})| + 1 & \text{if } s_{2i} = s_{2j-1} \\ 0 & \text{if } s_{2i} \neq s_{2j-1} \end{cases}$$

- 擬似Suffix linkからなる木を作成
  - LCAが要
  - 正しい木ではsuffix link と同じ
  - DFSでチェックしていく



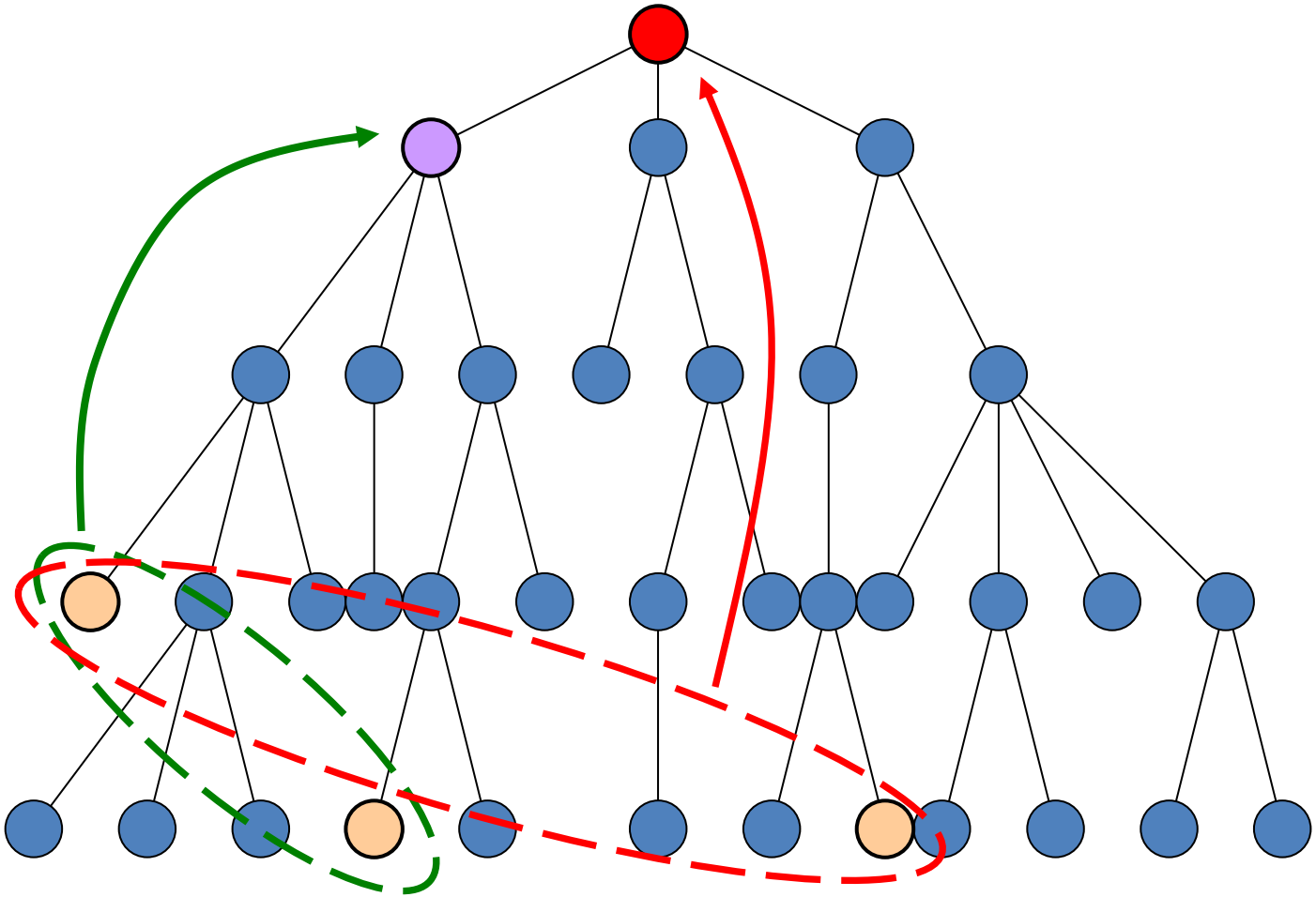
強力ツール： LCP

- ある接尾辞と別の接尾辞の最大の共通接頭辞(LCP=longest common prefix)長を求めることが、なんと線形時間の前処理をするだけで、$O(1)$でできる！
  - 接尾辞木→LCA (lowest common ancestor)
  - Farachのアルゴリズムに限らず、極めて多くの組み合わせパタンマッチングアルゴリズムに応用されている



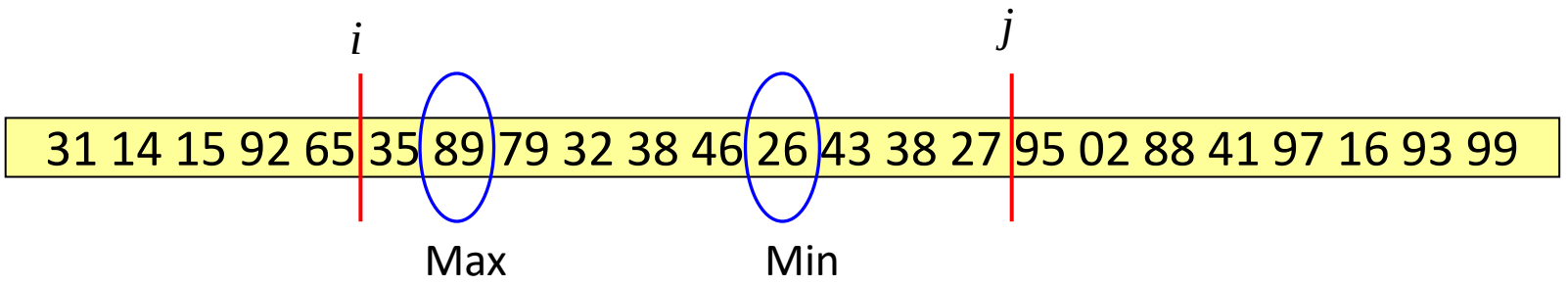
LCA

Lowest Common Anscestor



RMQ

- Range Maximum (Minimum) Query
  - 配列中のある領域の中で、最大（最小）の値を持つインデックスを探す

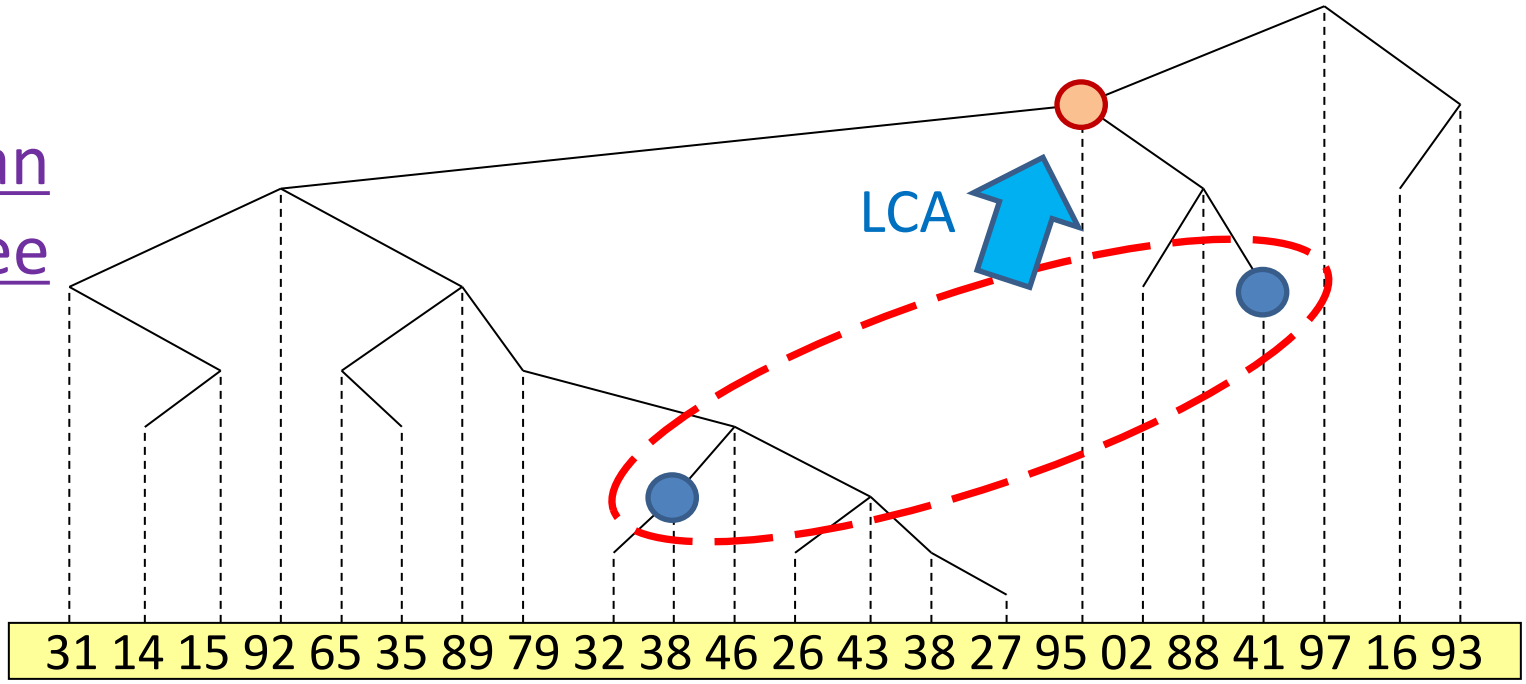


LCA & RMQ by Bender & Farach (1)

[Bender, Farach 2000]

- LCA→RMQ
  - Cartesian treeを作成すれば、RMQはLCAで解ける
- Cartesian tree
  - 数列から作成する （長さn の数列に対し、ノード数nの木を作成）
  - 各ノードは入力数列中の数に対応
  - 子ノードの値は親ノードの値以下
  - 数列中の左右の順番を木の中でも保存する

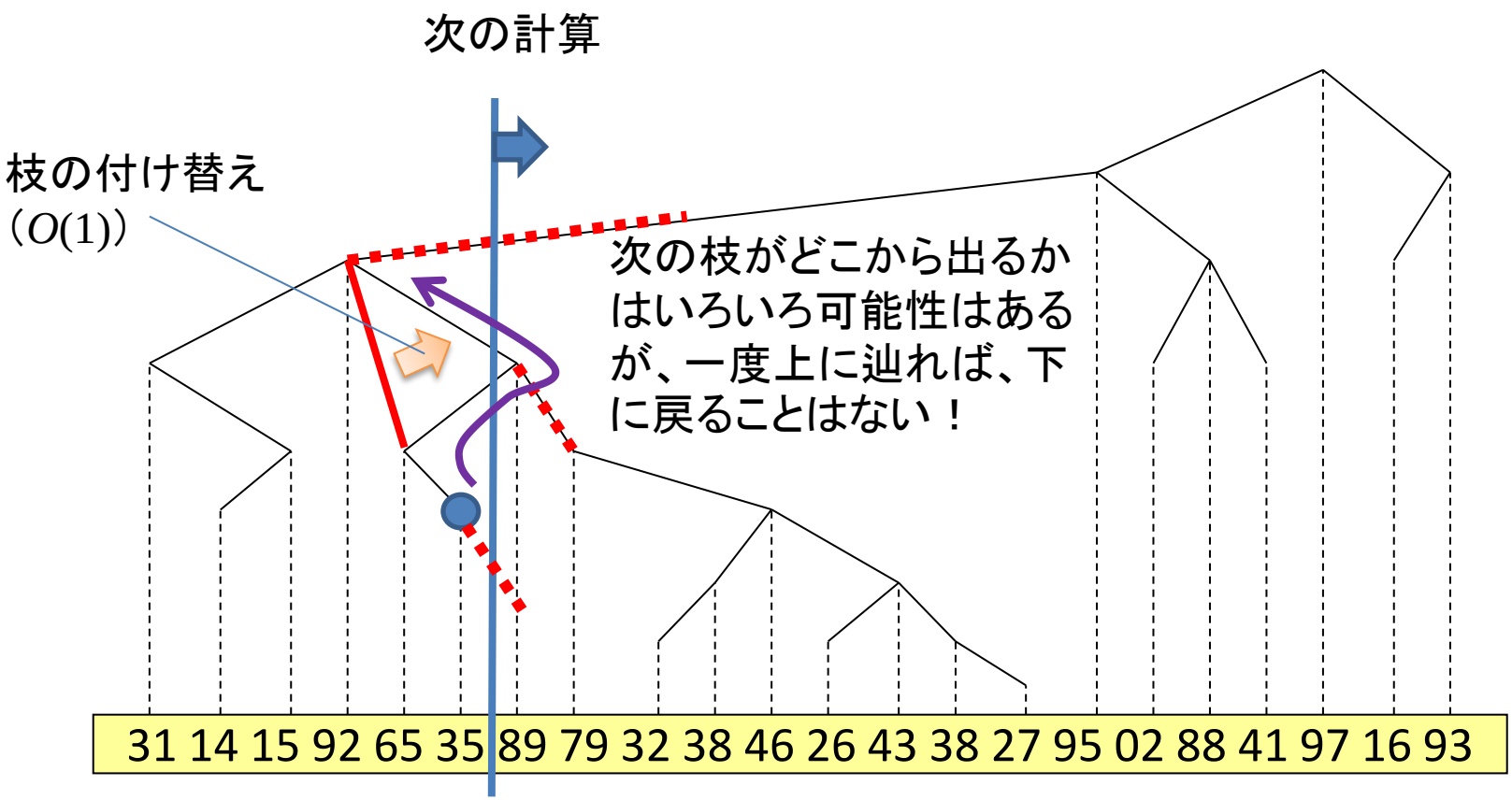
Cartesian Tree



LCA & RMQ by Bender & Farach (2)

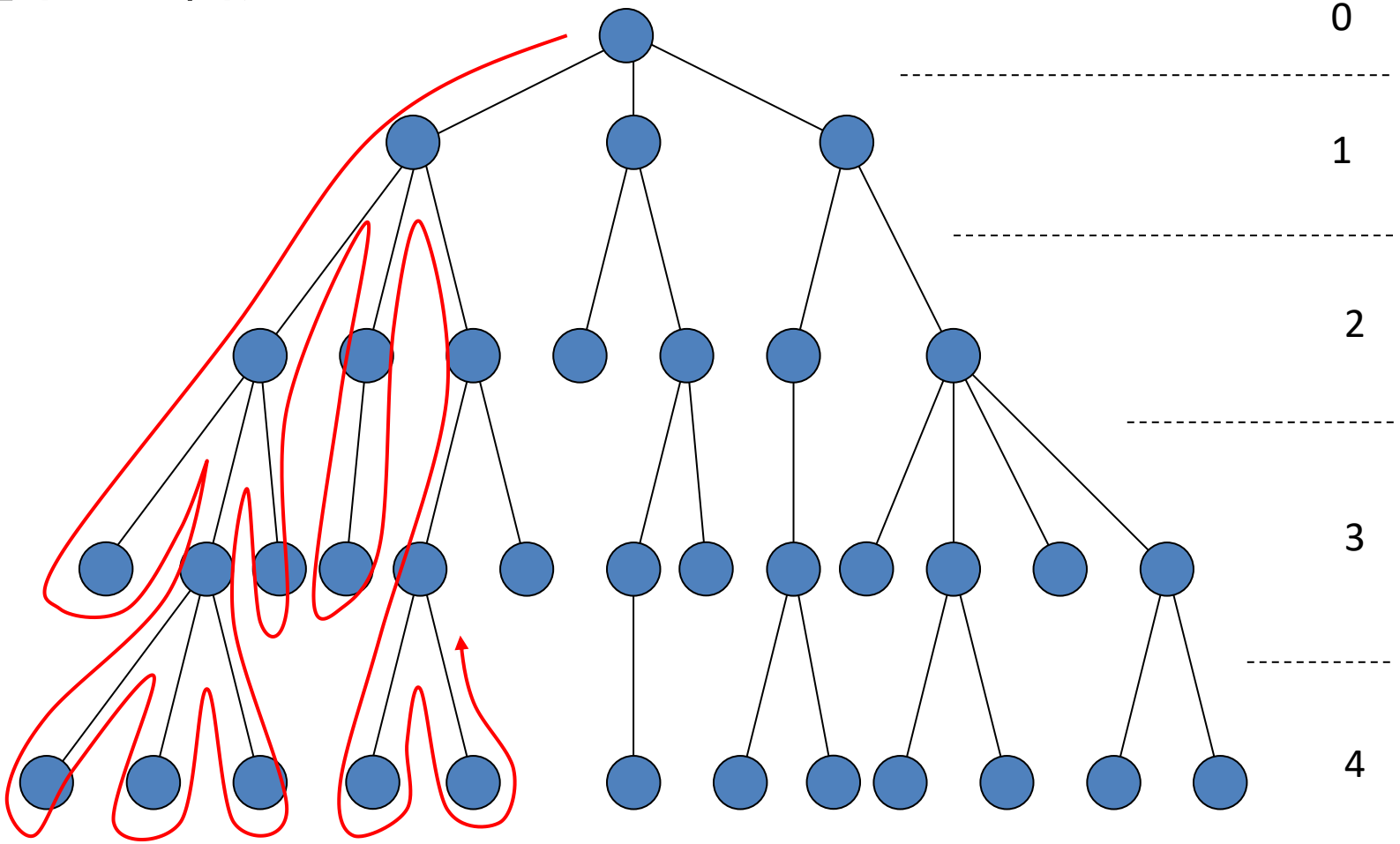
Cartesian treeの作成方法

- 単純に左からgreedyに作成すれば線形時間で構築可能
  - 木をトラバースする時間＝線形時間ですむため



LCA & RMQ by Bender & Farach (3)

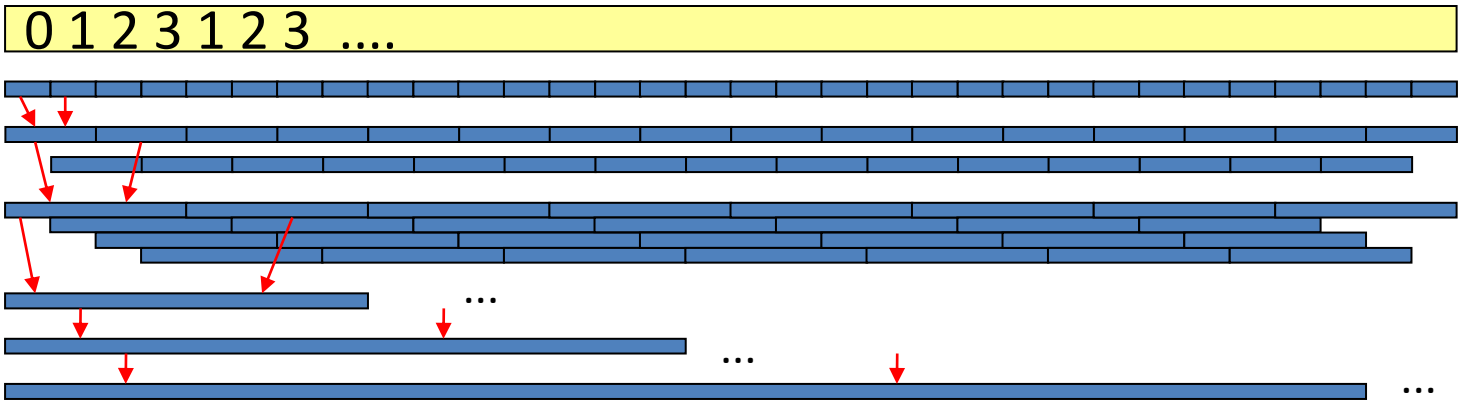
- ±1RMQ→LCA
  - ±1RMQとは値の変化が1のみである配列に対してRMQを求める問題



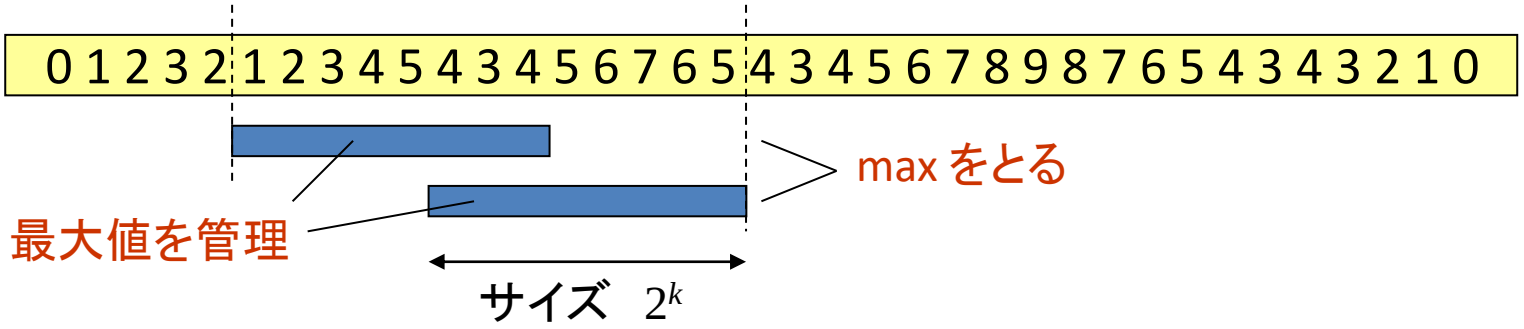
LCA & RMQ by Bender & Farach (4)

- $O(n \log n)$−preprocessing for $O(1)$ RMQ
  - $0<k\leq\log_2 n$ なる k すべてに対しサイズが 2^k のすべてのウィンドウにおいて最大（小）値を管理する
    - これは$O(n \log n)$で可能

前処理

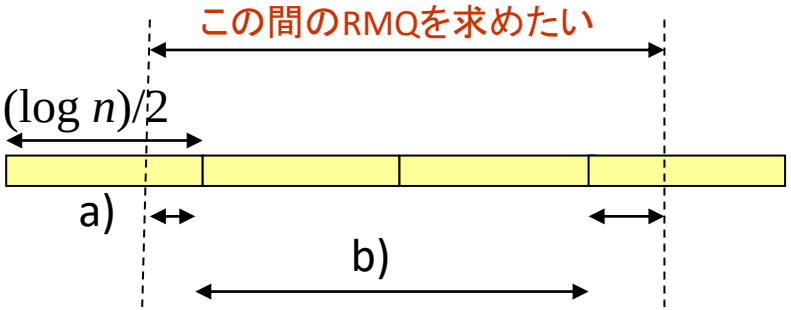


RMQ

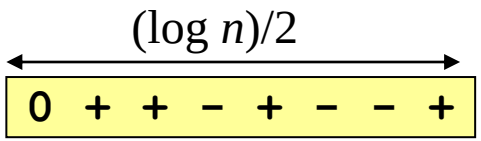


LCA & RMQ by Bender & Farach (5)

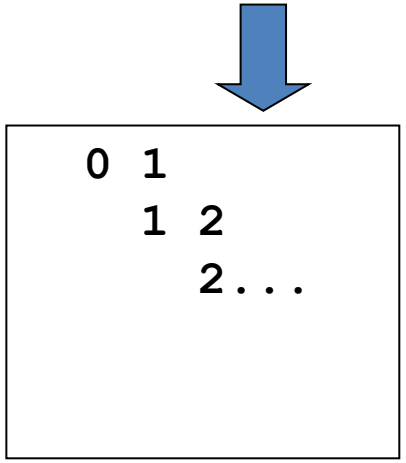
- 前処理時間$O(n)$の±1RMQ解法
  - $(\log n)/2$以下のサイズのブロックは $n^{1/2}$ の種類しかない！
    - 先頭の値を引いたものを考える
    - それぞれに対しナイーブなテーブルを作成しても$o(n)$
  - 重複するものを作らないようにすれば、全体で$O(n)$となる。



a) ブロック内RMQ
→　前処理は$o(n)$
b) は$2n/\log n$ 個の数列に対するRMQ
→　前処理は$O(n)$



先頭の値が変わっても、最大値の位置は不変



[*i*..*j*]の間の最大値の位置を示すテーブル

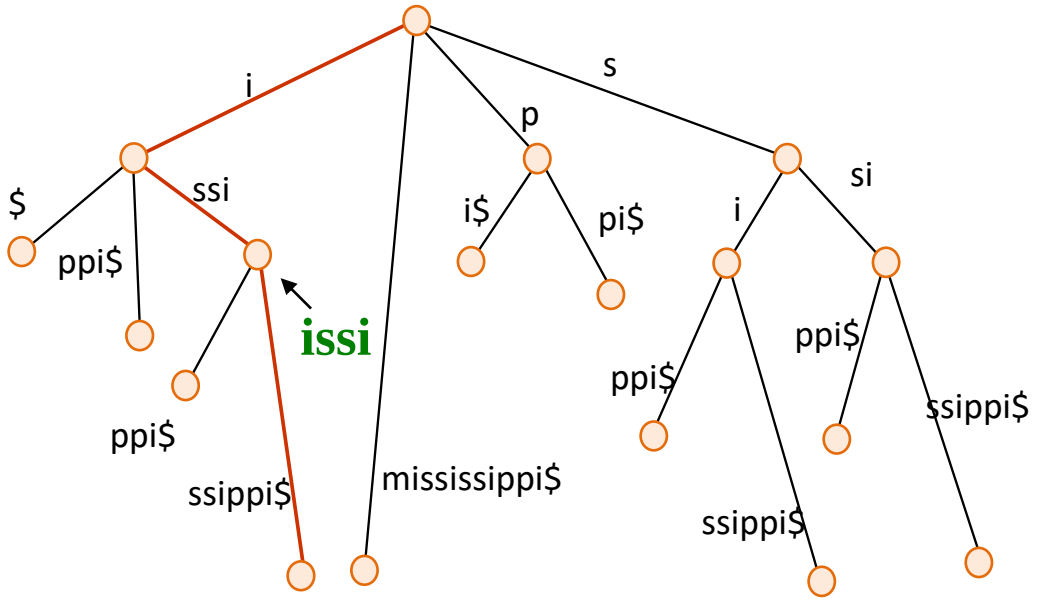
これまでの復習：接尾辞木とは

- 文字列 S のすべての接尾辞を表した trie
  - 枝のラベル　⇔　S の部分文字列
  - ルートから葉までのラベルを連結したもの　⇔　S の接尾辞
  - 線形サイズ・線形時間で構成可能

Suffix tree of 'mississippi$'

All the suffixes

mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$



接尾辞配列とは

- 接尾辞配列
  - すべての接尾辞のインデックスを辞書順にソートした配列
    - 辞書と同様、二分探索で任意の接尾辞、あるいはその接頭辞である部分文字列を探すことができる
  - 接尾辞木よりも遥かにコンパクト
    - 接尾辞木は $14n$ byte
    - 接尾辞配列は $5n$ byte
      - 整数・ポインタ等が32bit（4byte）、文字が8bit（1byte）で表現できる場合

すべての接尾辞

```
 0: mississippi$
 1: ississippi$
 2: ssissippi$
 3: sissippi$
 4: issippi$
 5: ssippi$
 6: sippi$
 7: ippi$
 8: ppi$
 9: pi$
10: i$
```

ソート

```
10: i$
 7: ippi$
 4: issippi$
 1: ississippi$
 0: mississippi$
 9: pi$
 8: ppi$
 6: sippi$
 3: sissippi$
 5: ssippi$
 2: ssissippi$
```

Suffix Array

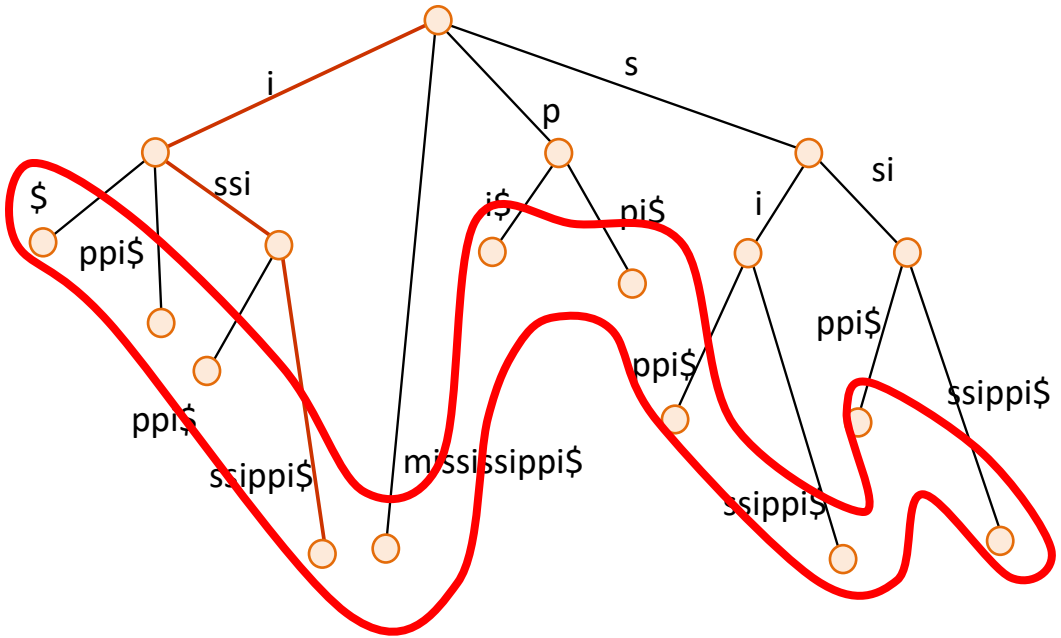
接尾辞配列と接尾辞木の関係

- 子供がソートされた接尾辞木における葉を並べたものと同じ
  - ただしそれぞれのノードの子供はアルファベット順にソートされているものとする
  - したがって接尾辞配列は接尾辞木から自明に線形時間で作成可能
    - 実は逆も可能（Kasai　et al. 2001・後述）
  - ＋αのデータ構造を用いるなどすれば、接尾辞木とほぼ同様の操作が可能となる場合が多い
    - 遅くなる場合もあるが、単純なデータ構造であるため、逆に速くなるアプリケーションも多い
    - アルゴリズムを考えるときは接尾辞木で、実装は接尾辞配列で、というのがよくあるパタン

Suffix tree of 'mississippi$'

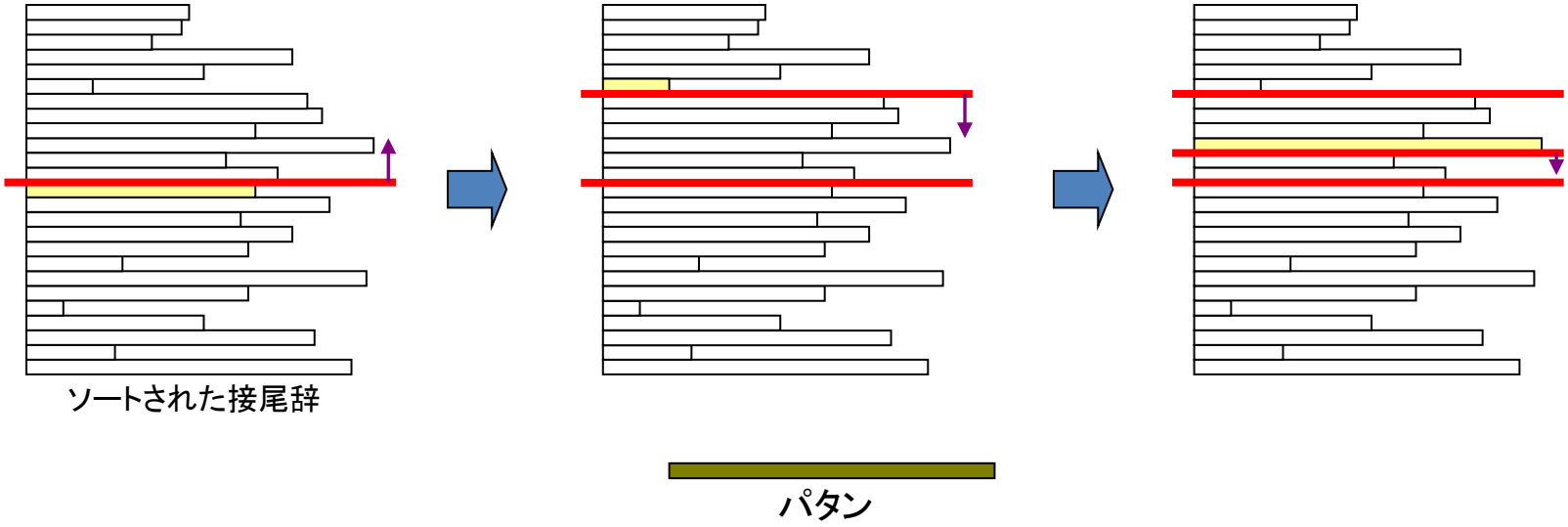
All the suffixes

mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$



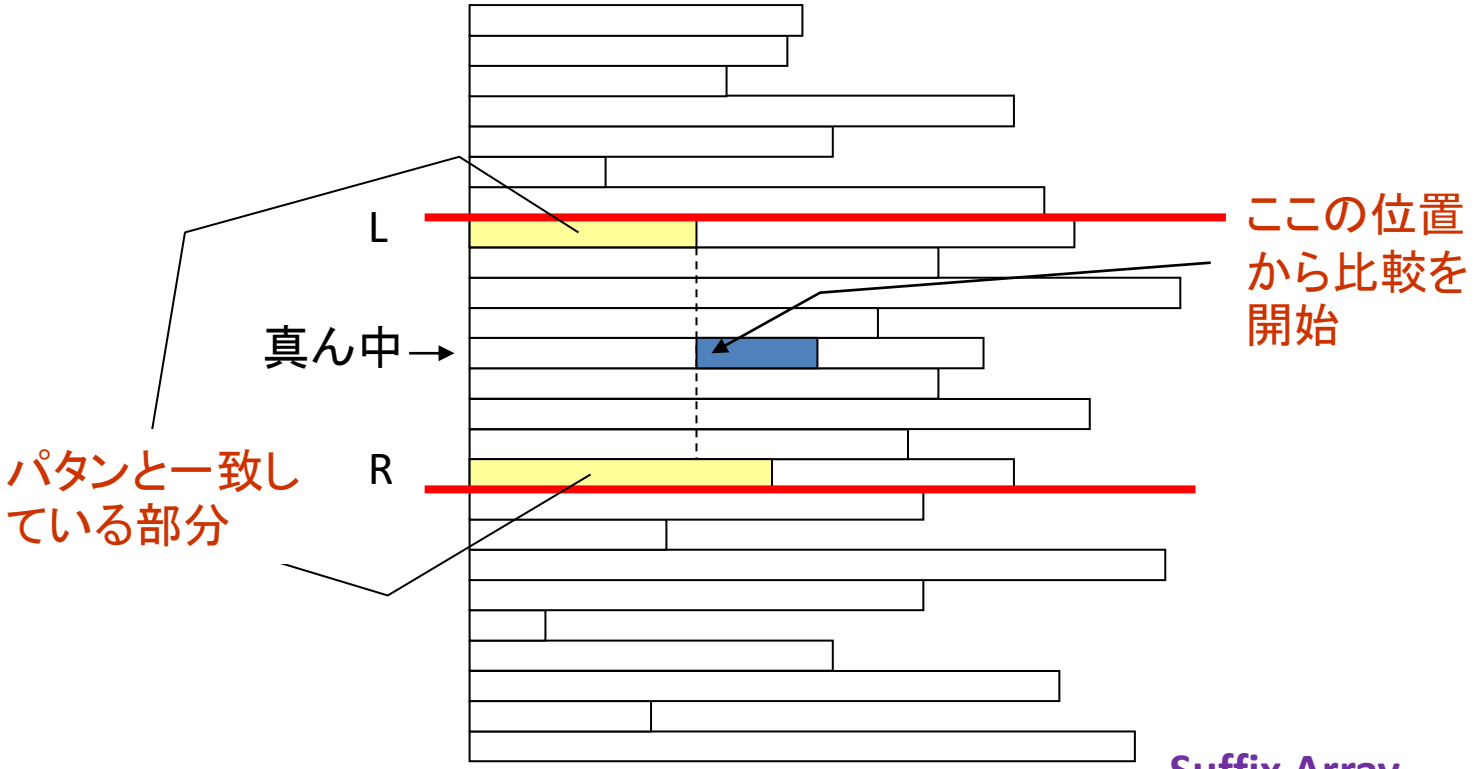
接尾辞配列を用いた検索（1）

- 二分探索
  - ソートされた数列を探索するのと異なり、長さ分調べる必要がある。
  - $O(m \log n)$ （n:テキストサイズ　m:パタンサイズ）
  - 平均的には $O(m + \log n)$
    - テキスト、パタンがランダムである場合



接尾辞配列を用いた検索 （2）

- パタンとのlongest common prefix length を用いた2分探索
  - 探索順序は変わらないが、文字列比較を減らすことで、少し速くできる。
    - ただし計算量は変わらない。
  - 追加のデータ構造を用いれば、最悪でも $O(m+\log n)$ や $O(m)$ とすることも可能（後述）
    - パタンを後ろから探索する方法などもある



例

“ipp”を検索

③ 10: i$ “ippi”と③のLCP長は 1

④ 7: ippi$ ← 発見

② 4: issippi$ “ippi”と②のLCP長は 1

1: ississippi$

0: mississippi$

① 9: pi$

8: ppi$ “ippi”と①のLCP長は 0

6: sippi$

3: sissippi$

5: ssippi$

2: ssissippi$

“mississippi”に対する接尾辞配列

②と③のチェックが終わった後は、チェックしないでも②と③の間のすべての接尾辞（ここでは④のみだが）は、その1文字目が‘i’であることがわかる！

接尾辞配列の構成アルゴリズム

- Naively from suffix trees
  - 常に$O(n)$だが、構築時にメモリが多く必要　（何のための接尾辞配列か？）
  - アルゴリズムが複雑なため現実には遅い
- Ternary quick sort
  - Bentley & Sedgewick '97
  - 一般的な辞書のソート用のアルゴリズム
  - ランダムな配列に対しては高速（$O(n \log n)$）
  - 繰り返しの多い塩基配列などでは遅く、$O(n^2)$になるケースも多い
- Doubling algorithm
  - Manber & Myers '92, Larsson and Sadakane '98.
  - 常に$O(n \log n)$
- Copy-based (BWT-like) algorithm
  - Itoh-Tanaka '99, Seward '00, Manzini-Ferragina '02, Schürmann and Stoye '05.
  - 省メモリ。最悪計算量は$O(n^2 \log n)$だが、実験には結構速い。
- Divide and merge algorithm　　（Farachのアルゴリズムのアイディアが源流）
  - Kärkkäinen & Sanders '03, Ko & Aluru '03, Kim et al. '03, Hon et al. '03, Na '05, Nong et al '09(←最速！).
    - 常に$O(n)$
  - Burkhardt & Kärkkäinen '03
    - $O(n \log n)$、作業メモリが$o(n)$

数列のソートとの関連

- 接尾辞配列は接尾辞のソート
  - 数列のソートの問題の一般化ともいえる
    - 同じ数字を含まない数列では、一般的なソートの問題と接尾辞配列を構成する問題は同じ問題
    - したがって、**一般的な数列に対する**接尾辞配列は**大小比較のみでは**$O(n \log n)$より速く作ることはできない！
    - $O(n)$を達成しているアルゴリズムは、integer alphabetを含め、**アルファベットサイズが小さい**、という事実を用いている
      - 一般的な数列ではalphabet sizeは（ほぼ）無限大
  - 数列のソートアルゴリズムを内部で利用することが多い
- どんなソートアルゴリズムを使っているか？
  - クイックソートを使っているもの
    - Bentley–Sedgewick, Larsson–Sadakane, etc.
  - ラディックスソートを使っているもの
    - Manber–Myers, Kärkkäinen–Sanders, Ko–Aluru, Larsson–Sadakane, etc.
  - マージソートを使っているもの
    - Kärkkäinen–Sanders, Ko–Aluru, etc.

大小比較による数列・接尾辞のソートの下限

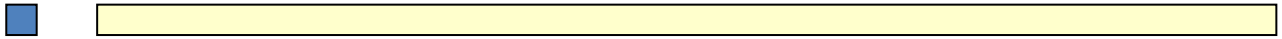
- 解の種類（数列のソートでも接尾辞のソートでも同じ）
  - $n!$
- 大小比較
  - Yes/Noの2種類の回答を得ることが可能
- 大小比較に基づく、いかなる入力でも処理可能なアルゴリズムの最大ステップ数の下限
  - 最良の場合には、これよりも少ないステップ数で求められることはもちろんあり得る（*e.g.* ソート済配列に対するバブルソートは線形時間）

$$
\begin{aligned}
t \geq \log n! &= \sum_{k=1}^{n} \log k \\
&> \int_{1}^{n} \log x \; dx \\
&= \Omega(n \log n)
\end{aligned}
$$

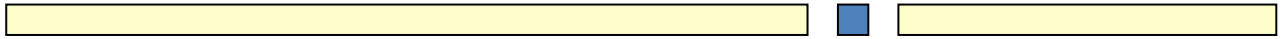
クイックソート

- 基本的な数列のソートアルゴリズム
  - 最悪$O(n^2)$、平均$O(n \log n)$

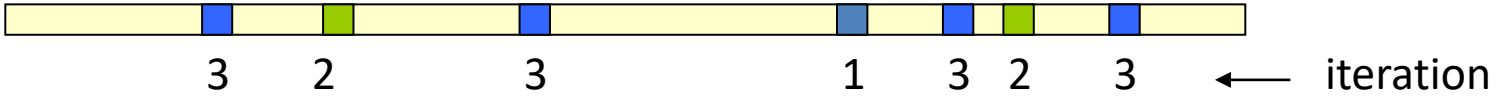
1. 一つpivotを選ぶ



2. それより大きいものと小さい（あるいは同じ）ものに分ける

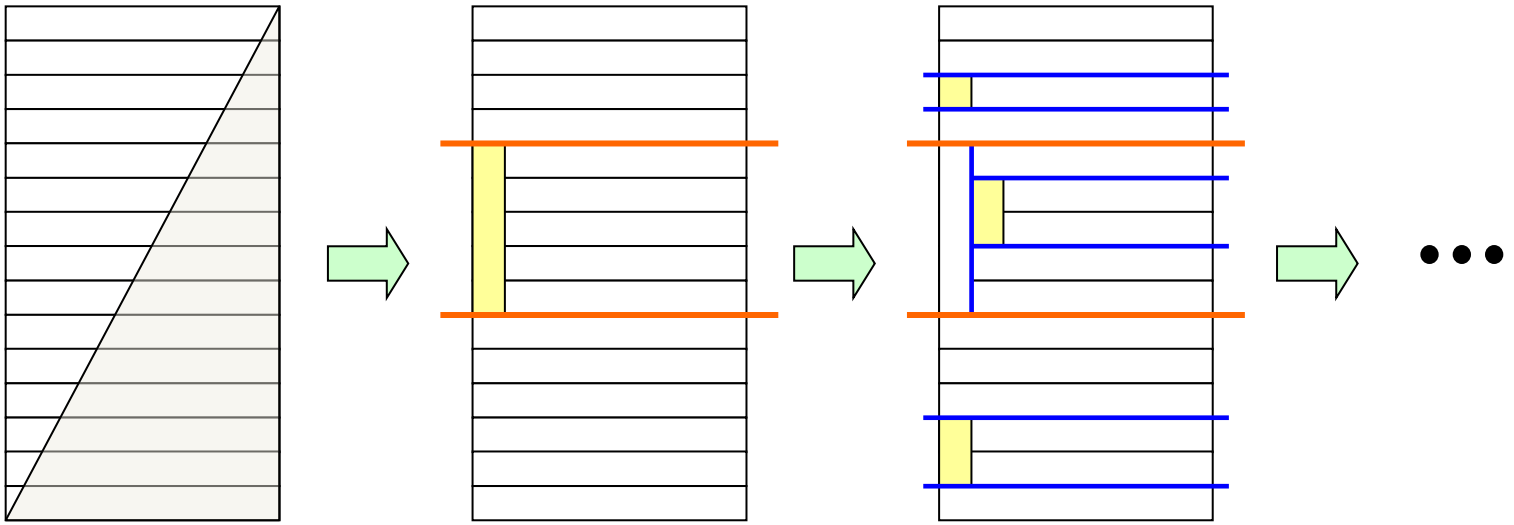


3. 大きいもの、小さいものに対して再帰的に同じことを行い、最終的にソートされたら終了する



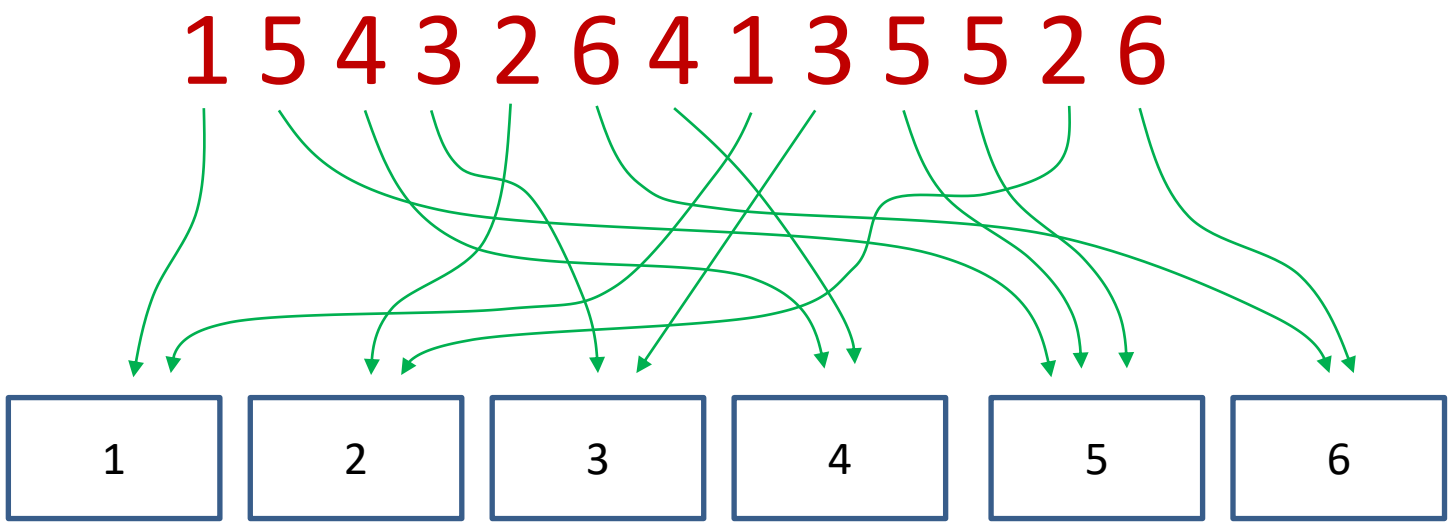
Bentley & Sedgewick '97

- Ternary quick sort
  - クイックソートを文字列用にチューニングしたもの
  - インプリは最も簡単
  - ランダムなら$O(n \log n)$だが、通常の文字列のソートの場合よりも遅い$O(n^2)$になるケースも多い
    - 2-wayでも計算量は変わらないが、インプリするとこちらの方が速い
    - DNAなどでは、「NNNNN」とか「ATATATATA」などの繰り返しがあったりするだけでかなり危険
    - 一般的な文章でも「わたし」の次には「は」が来ることが多い
      - いずれもうまく等分割とならない原因となる



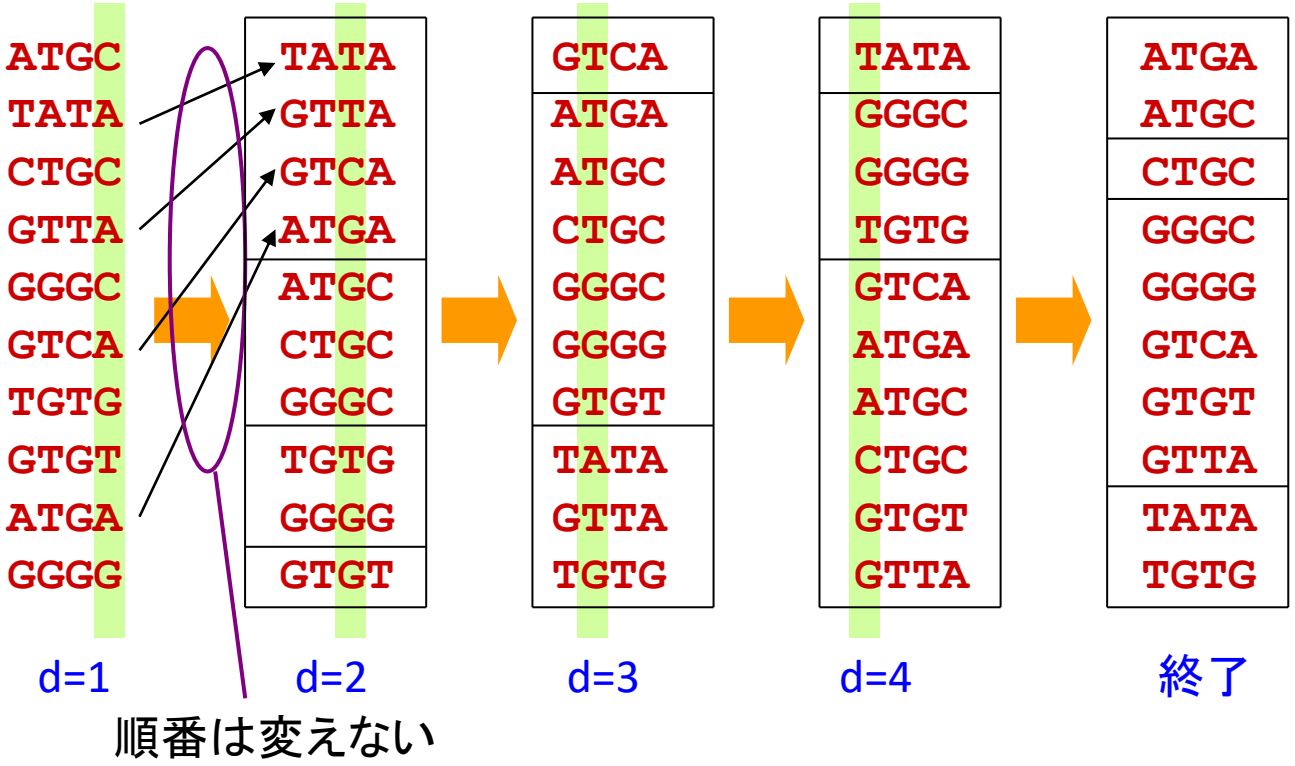
大小比較以外も使うことができればもっと計算量が下がる可能性がある

- バケツソート
  - 最大値 c の n 個の非負整数のソートは$O(c+n)$で可能
  - 数字を用意した箱に入れるだけ！



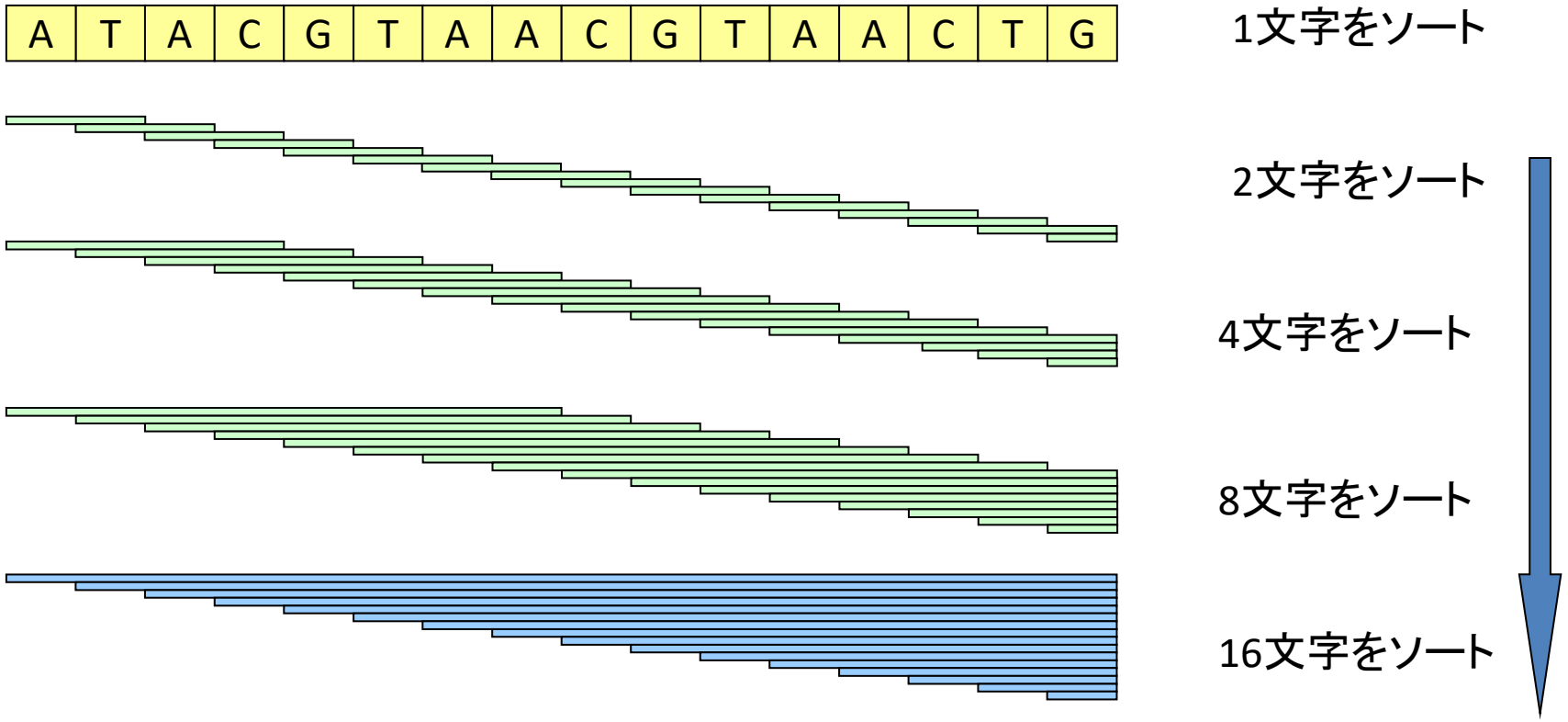
ラディックス・ソート

- ラディックス（基数）・ソート
  - 下の位（あるいは文字）からソートして行く
    - その桁の文字が同じものについては順番を変えないようにする
    - バケツソートで $O(n+s)$　（sはアルファベットサイズ）
  - 計算量　$O((n+s)b)$　（bは桁数）



Manber & Myers '92 (1)

- Doubling algorithm
  - $\log n$ 回のラディックス・ソートで計算
    - ソートするたびに2文字を1文字（ソートされた順番）に置き換える
    - それぞれの段階は線形時間で可能！ -> $O(n \log n)$



Manber & Myers '92 (2)

ラディックス・ソートの適用のしかた

最初の文字列（i=0）

3 0 4 1 5 9 2 3 5 2 3 1 4 5 2 3 1 2

となりあう2文字でソートした時のソート順での番号に置き換える

04 ▸ 0	23 ▸ 5	45 ▸ a
12 ▸ 1	30 ▸ 6	52 ▸ b
14 ▸ 2	31 ▸ 7	59 ▸ c
15 ▸ 3	35 ▸ 8	92 ▸ d
2$ ▸ 4	41 ▸ 9	

文字はinteger

このソートは2桁の基数ソートで線形時間で可能

6 0 9 3 c d 5 8 b 5 7 2 a b 5 7 1 4

次はとなりあう文字ではなく2つ隣の文字とあわせて基数ソートを行う（i番目では　2^i だけ隣とあわせる）

全部計算するか、アルファベットサイズと文字列長が等しくなったら終了

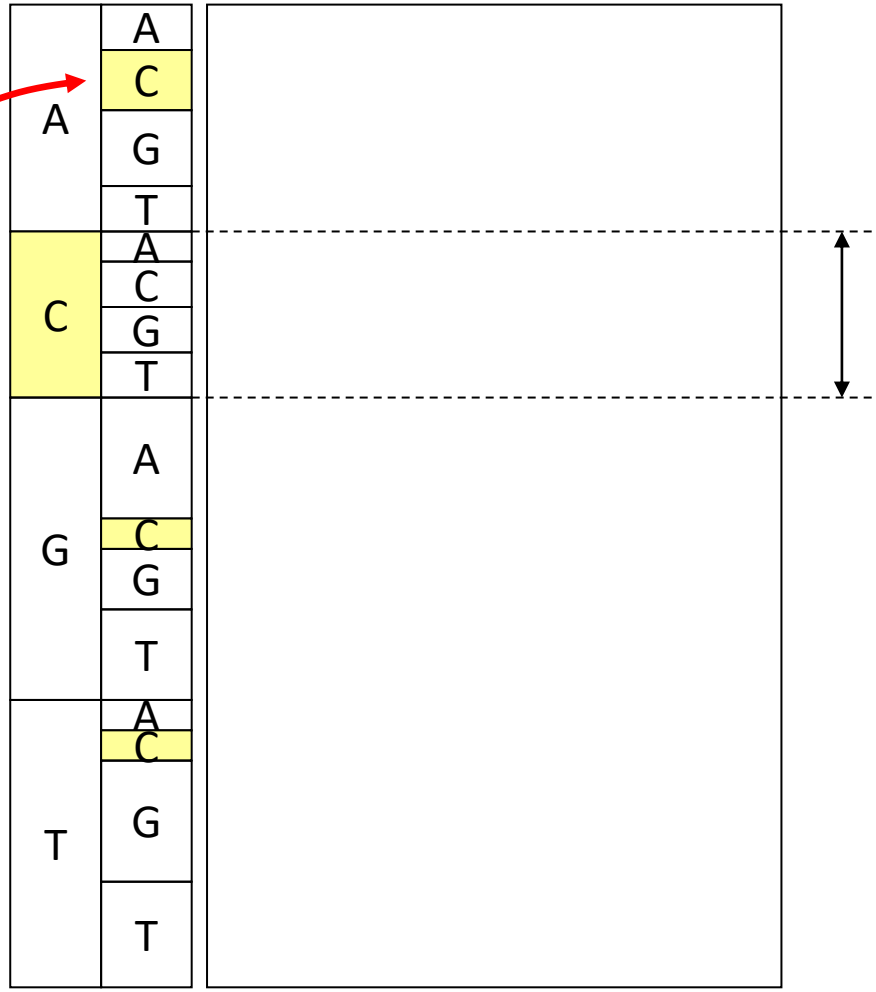
Larsson & Sadakane '98

- Manber & Myersの問題点
  - 反復の最後の方では、ほとんど順番は不変
    - それほど長い一致が存在するわけではない
- Larsson & Sadakane Algorithm
  - Ternary quick sortを用いてダブリングを行う
    - 1文字目が同じであるグループの中で、2文字目をTernary sort
      - このような（文字列ではない）文字に対するTernary sortは Ternary split quick sort あるいは Bently–McIlroyアルゴリズムとよばれる
    - ソートが終了した場所はスキップ
      - これができるため、M&Mより相当速い
    - 一見 $O(n\,(\log n)^2)$ っぽいが、実は $O(n \log n)$
      - すなわちM&Mと同じ

Seward "Copy" Algorithm '00（概略）

- SA中に似た並びがあることを利用！
  - 平均LCP長が短い場合は結構速い
  - 必要メモリ量が小さい

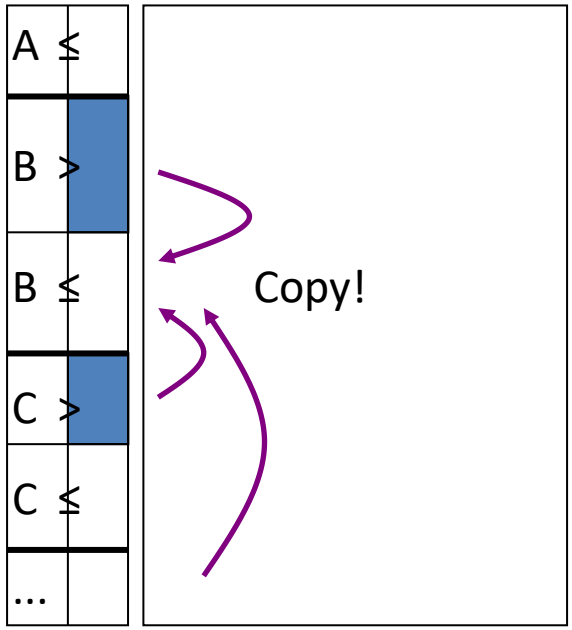
一つ前の文字がAのものに関して順番にコピー



1. まずは2文字だけでソート
2. 一番少ない文字を探す（Cだとする）
3. まずはCで始まるsuffixをすべてソートする。ただし、CCで始まるsuffixのソートはCA,CG,CTの結果を用いれば計算できる
4. ＊Cで始まるものに計算結果をコピーする
5. 次に少ないものを選んで、繰り返す（但し、もう計算できているものは計算しない）

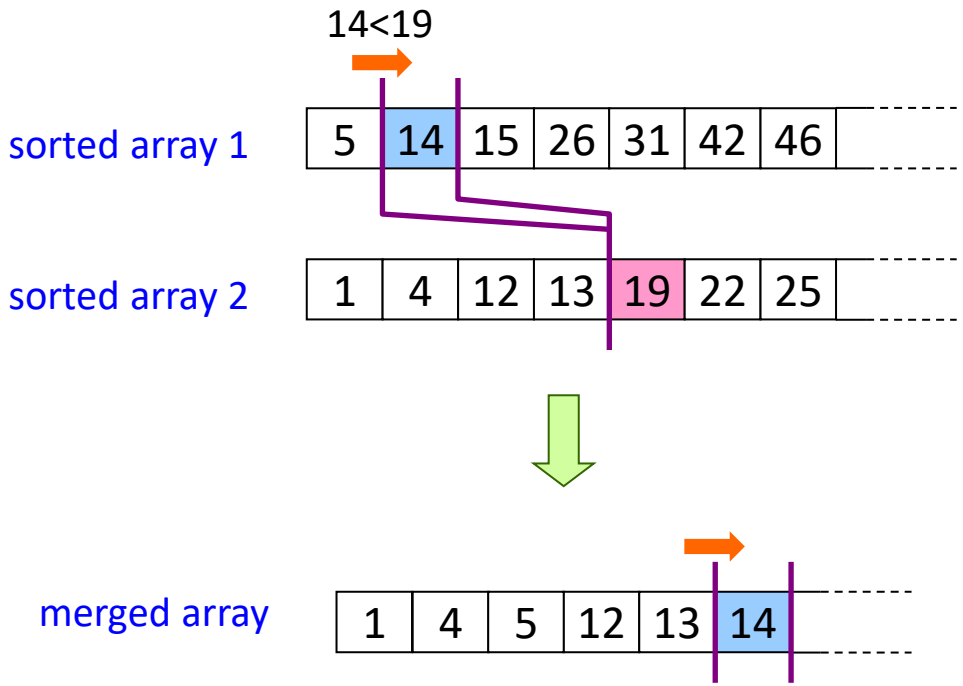
Itoh-Tanaka Algorithm '99

- 同様のCopy系アルゴリズム
  - 後の$O(n)$のKo-Aluru AlgorithmやInduced Sortingのアイディアの種となったアルゴリズム
- アルゴリズム
  - suffix を2つにわける
    - type 1：1文字目>2文字目　（BA....など）
    - type 2：1文字目≤2文字目　（AA... やAB...など）
  - Type 1を普通にソート（Ternary quick sort）
  - その結果を用いて全体をソート



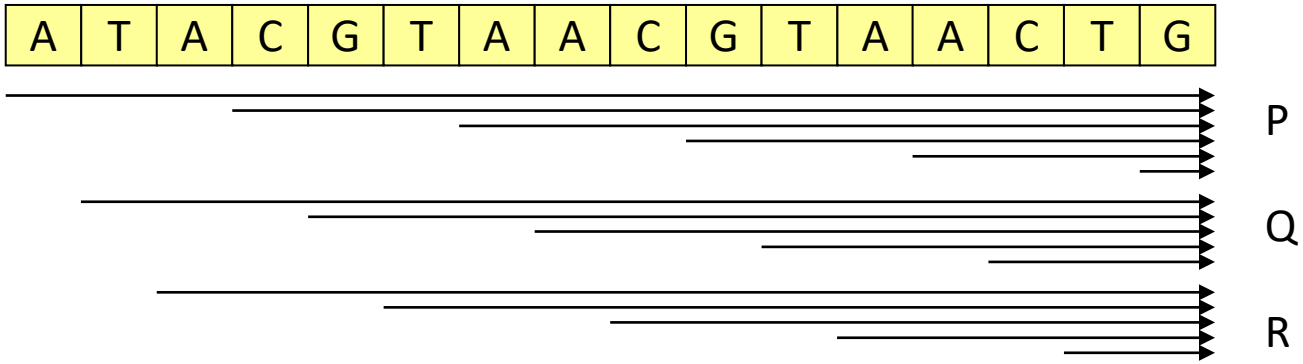
復習：Merge Sort

- 2つのソート済配列の併合をするには？
  - 2つの配列を端から順番に見て小さい方を出力
  - **数字の比較は$O(1)$でできる**ので全体で線形時間
    - ここが重要
    - 接尾辞は数字ではないので、接尾辞配列のマージは極めて難しい！
  - 数列のソートの場合には、マージを再帰的に適用することで（最悪でも）$O(n \log n)$で全体をソートできる



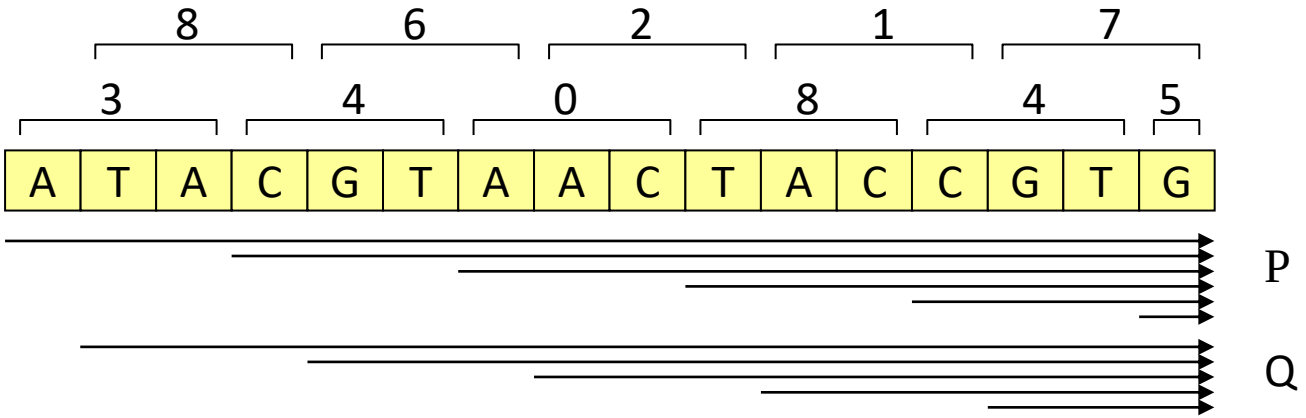
Kärkkäinen & Sanders '03 (1)

- Farachの接尾辞木のアルゴリズムを洗練させたアルゴリズム
- 接尾辞を3種類に分ける
  - それぞれ$3i$, $3i+1$, $3i+2$ 番目から始まる接尾辞集合をそれぞれP, Q, Rとする
  - 集合P+Qに含まれる接尾辞のみからなる接尾辞配列を作成する
    - $f(2n/3)$ 時間で再帰的に計算することが可能
      - $f(n)$: 全体の計算時間
  - その接尾辞配列を用いて接尾辞集合Rに含まれる接尾辞のみからなる接尾辞配列を作成する
    - $O(n)$ 時間
  - 2つの配列をマージする（逆接尾辞配列をうまく用いる）
    - $O(n)$ 時間



Kärkkäinen & Sanders '03 (2)

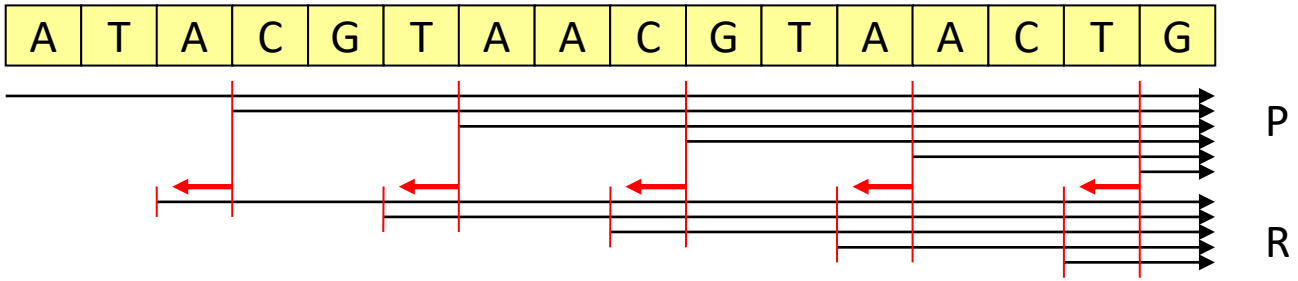
- 接尾辞集合P＋Qに対する接尾辞配列の作り方
  - P+Qに含まれる各接尾辞の先頭3文字の組をまずソート
    - 基数ソートで線形時間
  - ソート順の番号に3文字の組を変更したP,Qに対応する文字列それぞれ作る
    - 2つの文字列を連結し（間に終端記号をはさむ）、それに対して接尾辞配列を計算する（このアルゴリズム全体を再帰的に適用）
    - 結果を用いてP＋Qに対する接尾辞配列を作成（自明）



340845$86217に対する接尾辞配列を作成する

Kärkkäinen & Sanders '03 (3)

- 部分接尾辞集合Rに対する接尾辞配列の作り方
  - Pに対する接尾辞配列から基数ソート一回で作ることができる！（線形時間）
  - PはP+Qの一部なので、先に作成したP+Qに対する接尾辞配列の中からPに対応するindexのみ取り出せばPに対する接尾辞配列が自明に作成可能（線形時間）



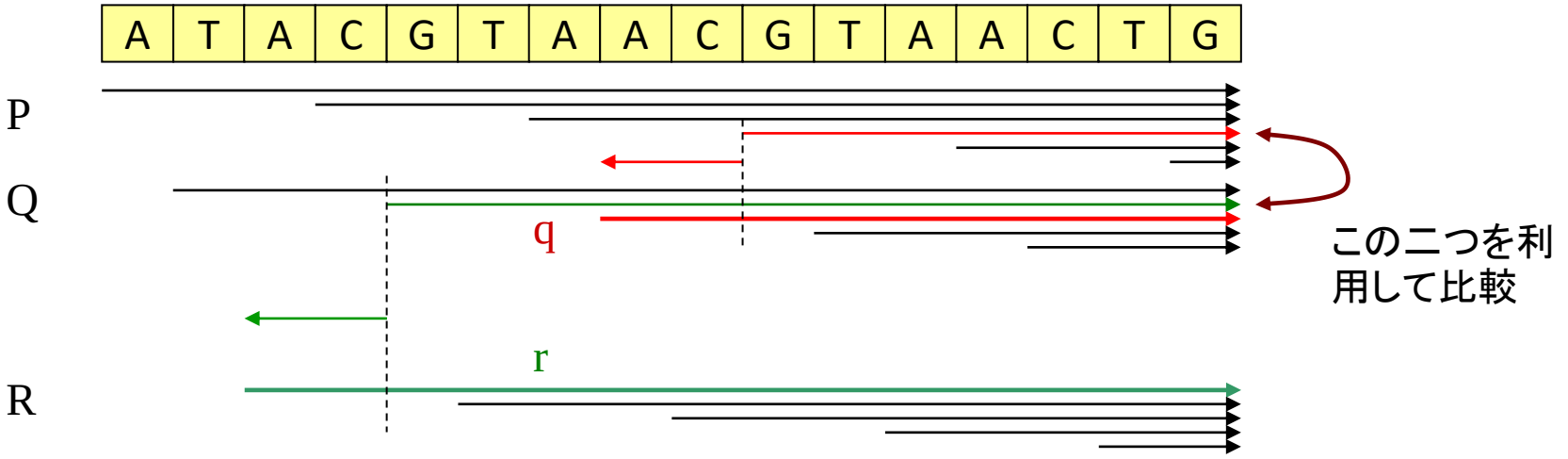
Kärkkäinen & Sanders '03 (4)

- 逆接尾辞配列
  - $SA^{-1}[i] = j \leftrightarrow SA[j] = i$
  - 様々な他のアルゴリズムでも使用されるデータ構造
  - 線形時間で作成可能

	Suffixes of theText			SA			SA^{-1}
0:	mississippi$		10:	i$			4
1:	ississippi$		7:	ippi$			3
2:	ssissippi$		4:	issippi$			10
3:	sissippi$		1:	ississippi$			8
4:	issippi$		0:	mississippi$			2
5:	ssippi$	ソート	9:	pi$	逆		9
6:	sippi$		8:	ppi$			7
7:	ippi$		6:	sippi$			1
8:	ppi$		3:	sissippi$			6
9:	pi$		5:	ssippi$			5
10:	i$		2:	ssissippi$			0

Kärkkäinen & Sanders '03 (5)

- 2つのマージをするには？
  - 接尾辞同士の比較が $O(1)$ でできれば、線形時間のマージができる（マージソートと同じ）
  - 逆接尾辞配列を用いる
    - 逆接尾辞配列を用いれば、P+Qに入っている接尾辞同士が$O(1)$で比較できる
      - それに加えてその直前1文字または2文字を比較すれば、P，R間あるいはP，Q間の比較ができる



Kärkkäinen & Sanders '03 (6)

- 計算時間
  - $f(n) = f(2n/3) + O(n)$
  - これは $O(n)$
- 一般的に　$f(n)=c_1 \cdot f(c_2 \cdot n)+O(n)$ $(0<c_2<1,\ 0<c=c_1c_2)$の時、
  - $0<c<1$ ならば $f(n) = O(n)$
  - $c=1$ ならば $f(n) = O(n \log n)$
  - $c>1$ ならば $f(n) = O(n^{-\log_{c_2} c_1})$

$$f(n) < c_1 \cdot f(c_2 \cdot n) + a \cdot n < {c_1}^2 \cdot f({c_2}^2 n) + a(1+c)n < \cdots < \underbrace{{c_1}^{\log n} \cdot f(\text{const})}_{O(n)} + \underbrace{a\left(1 + c + c^2 + \cdots + c^{\log n}\right)n}_{0<c<1 \text{ ならば } f(n) = O(n)}$$

原論文: Kärkkäinen, J., and Sanders, P. (2003) "Simple Linear Work Suffix Array Construction", *Proc. ICALP, LNCS 2719*, pp. 943-955.

Burkhardt & Kärkkäinen '03 (1)

- KSの（理論的）省メモリ化
  - $o(n)$ の追加空間計算量で $O(n \log n)$ 時間としたもの
- Difference Coverというものを使う
  - $[0..n-1]$ に対し $O(n^{1/2})$ の大きさのカバーを $O(n^{1/2})$ 時間で求めることができる　（より厳密には[Colbourn&Ling, '00]）

[0..99] のdifference coverの例

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90

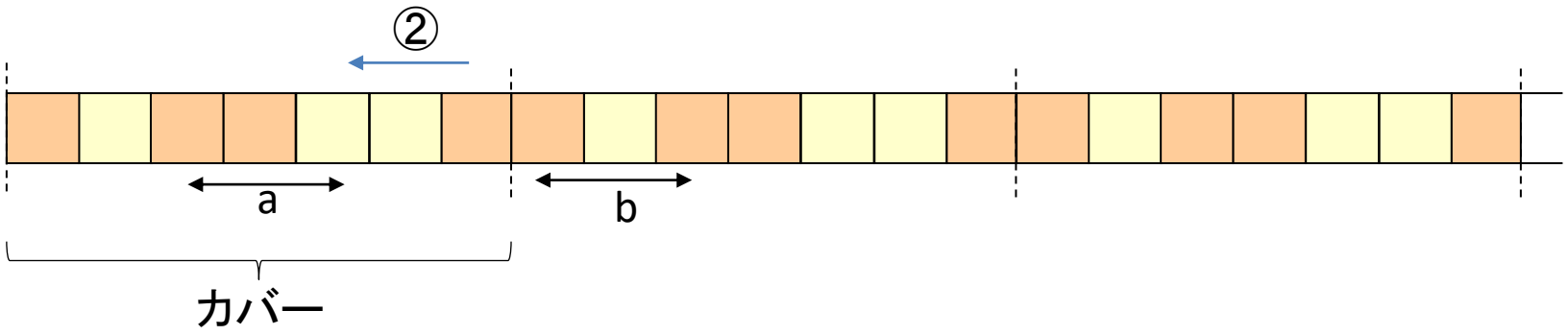
（これが簡単だがもう少しだけ賢いカバーも存在する）

$0 \sim 99$ を $(d_i - d_j) \bmod 100$ で表現できる

cf. 素数ものさし
577円（消費税込で623円）
@京大生協

Burkhardt & Kärkkäinen '03 (2)

- より大きな範囲でK&Sと同様の計算を行う



①  に対する接尾辞配列を作成する

②  に対する接尾辞配列をradix sortで作成する

③ それらをマージする

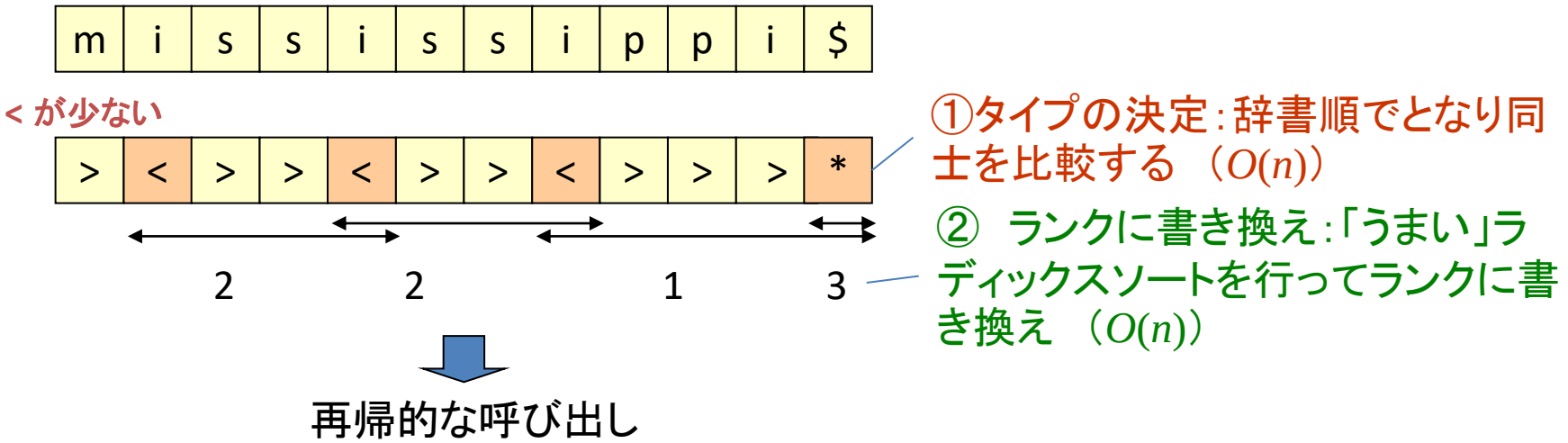
たとえば、aの比較は、先頭5文字の比較＋bの比較で行えば
O(カバーのサイズ)で計算可能

Burkhardt & Kärkkäinen '03 (3)

- Difference cover に入っている剰余に対応する接尾辞だけからなる接尾辞配列を作成
- 入っていないものに関してはそれぞれradix sortで作成可能
  - radix sortの回数はcoverの大きさに比例
- ウィンドウサイズがkならば、
  - この時カバーのサイズは $c \cdot k^{1/2}$
  - k回の比較演算で、どの違う2つの剰余に対応する接尾辞の辞書順比較が可能
  - すなわち $f(n) = f(c \cdot n/k^{1/2}) + O(kn)$
  - $k = \log n$ とすると $f(n) = O(n \log n)$
  - K＆Sは $k=3$ としたアルゴリズムに相当
    - $\{0,1\}$は$\{0,1,2\}$のカバー

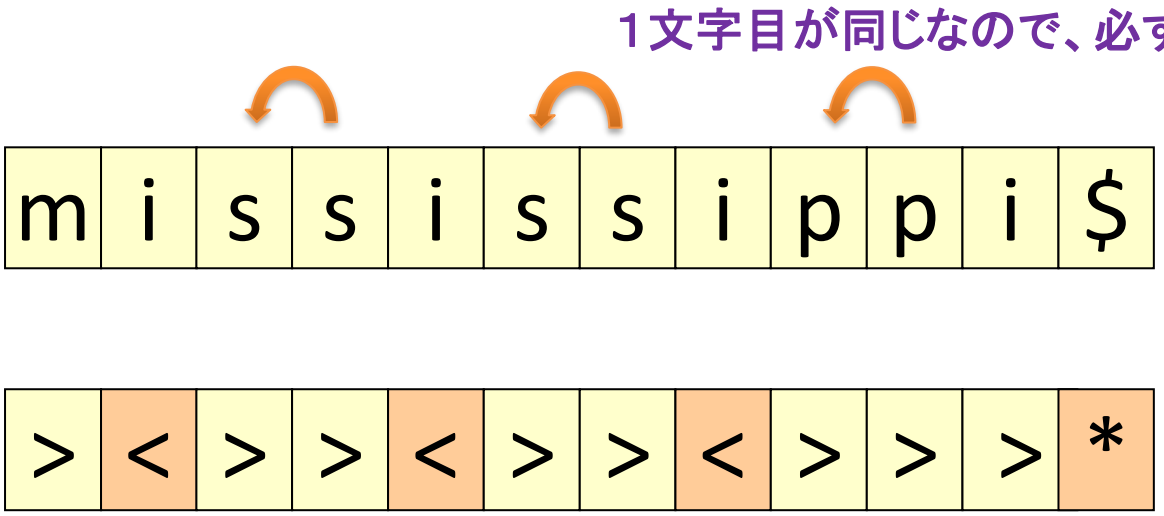
Ko and Aluru '03 (1)

- K&Sと同様のDivide and Merge 系アルゴリズム
  - となりの接尾辞との辞書順の大小で接尾辞を「＞」と「＜」の2つのタイプに分割し、それらのうち数の少ない方に対して接尾辞配列を作成
    - Itoh−Tanaka algorithmのアイディアにも類似
    - 「うまく」ラディックスソートでランキングすることで小さい問題に変換
  - その結果を利用して、計算していない接尾辞も同様の「うまい」ラディックスソートで計算
  - マージも容易
    - 同じ文字で始まる「>」タイプの接尾辞は「<」タイプの接尾辞より必ず辞書順で小さい　＝　そこから先は$O(1)$で比較可能



Ko and Aluru '03 (2)

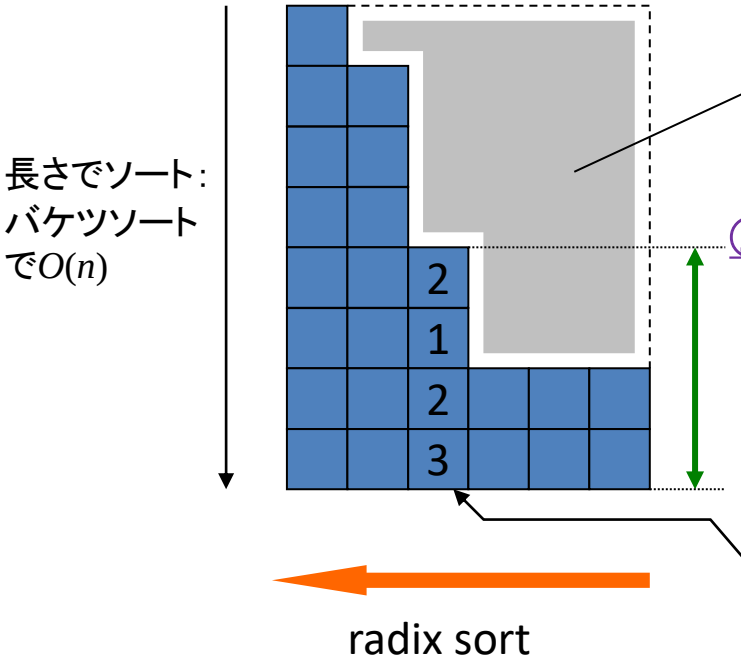
- 「<」タイプと「>」タイプの線形時間決定法
  - 後ろから見ていくだけ
    - 一つ後ろの接尾辞と先頭１文字だけチェックし、先頭文字が異なる場合
      - その大小で「＞」タイプか「＜」タイプかがO(1)でわかる
    - 先頭文字が同じ場合
      - 一つ後ろの接尾辞と同じタイプなので、やはりそれ以上チェックする必要はない！



Ko and Aluru '03 (3)

「うまい」radix sort　（このルーチンを2箇所で使う）

”<”文字列のソート



最後の文字は "<" タイプなので、そこまで同じ文字列は
ソート列の後ろに来ることになる
- 例えば "isp" > "ispi"
- ">"文字列のソートでは逆

②それから「かしこい」ラディックスソート

この幅がwならば$O(w)$で一文字分のソートが可能
＝全体で $O(N)$ でソートが可能

①まず「かしこく」ランキング

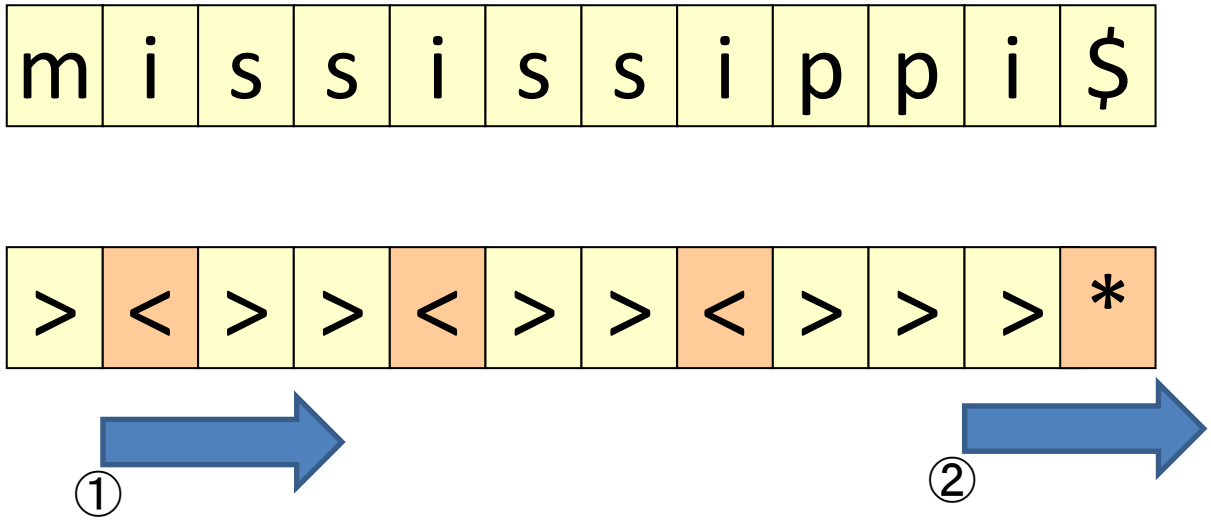
テーブル中の全文字の順位（ランク）を決定する
ことが全体で $O(n+N)$ のバケツソートで可能
Step 1: この表に含まれる全文字をソート
Step 2: そのランク結果を（縦の）各列に分配
さらに分配時に、その列でのランクに変換！

N（表のサイズ（表中の文字列の長さの総和））　≤　$1.5 \times n$（元の文字列の長さ）

ここのインプリはかなり速度を左右することに注意

Ko and Aluru '03 (4)

- マージ時の$O(1)$比較
  - ＜と＞の比較は1文字目のみでOK！

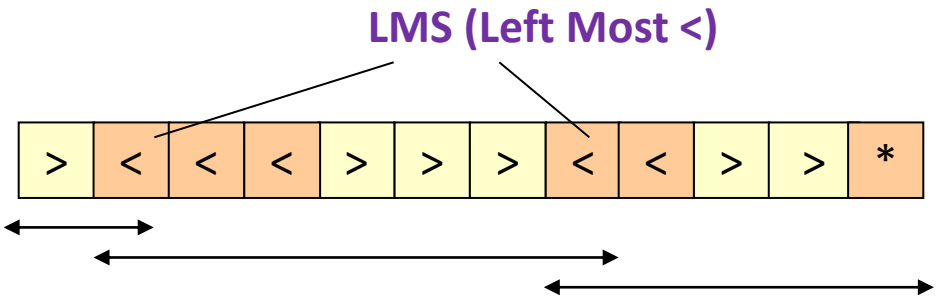


①と②は、どちらも‘i’で始まる接尾辞だが、異なるタイプなので、どちらの接尾辞が辞書順で速いかは2文字目以降を見ずとも決定可能！

Induced Sorting (1)

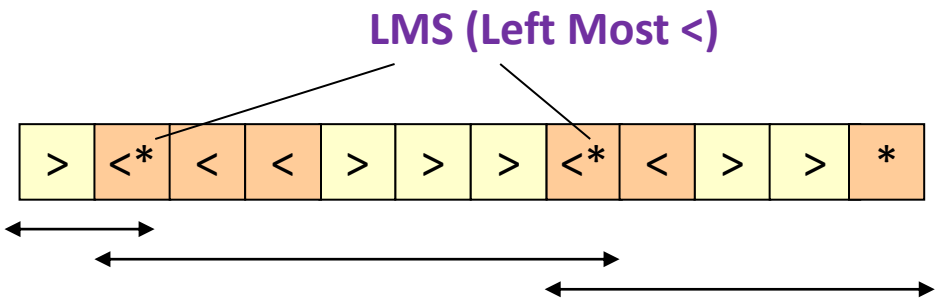
[Nong-Zhang-Chan '09]

- Ko-Aluru
  - Kärkkäinen-Sandersと比べて高速
    - KSと比べて部分問題サイズが小さいことによる
- Induced Sorting [Nong-Zhang-Chan ’09]
  - 接尾辞配列アルゴリズムの決定版
    - Ko-Aluru のアイディアの改良
  - 文字列の分割方法をさらに洗練させ、再帰的に解く「小さい部分問題」をより小さくしたことにより、再帰計算の時間が大幅に減少した
  - （論文によれば）KAより約1.5倍高速。メモリ量も2/3程度。

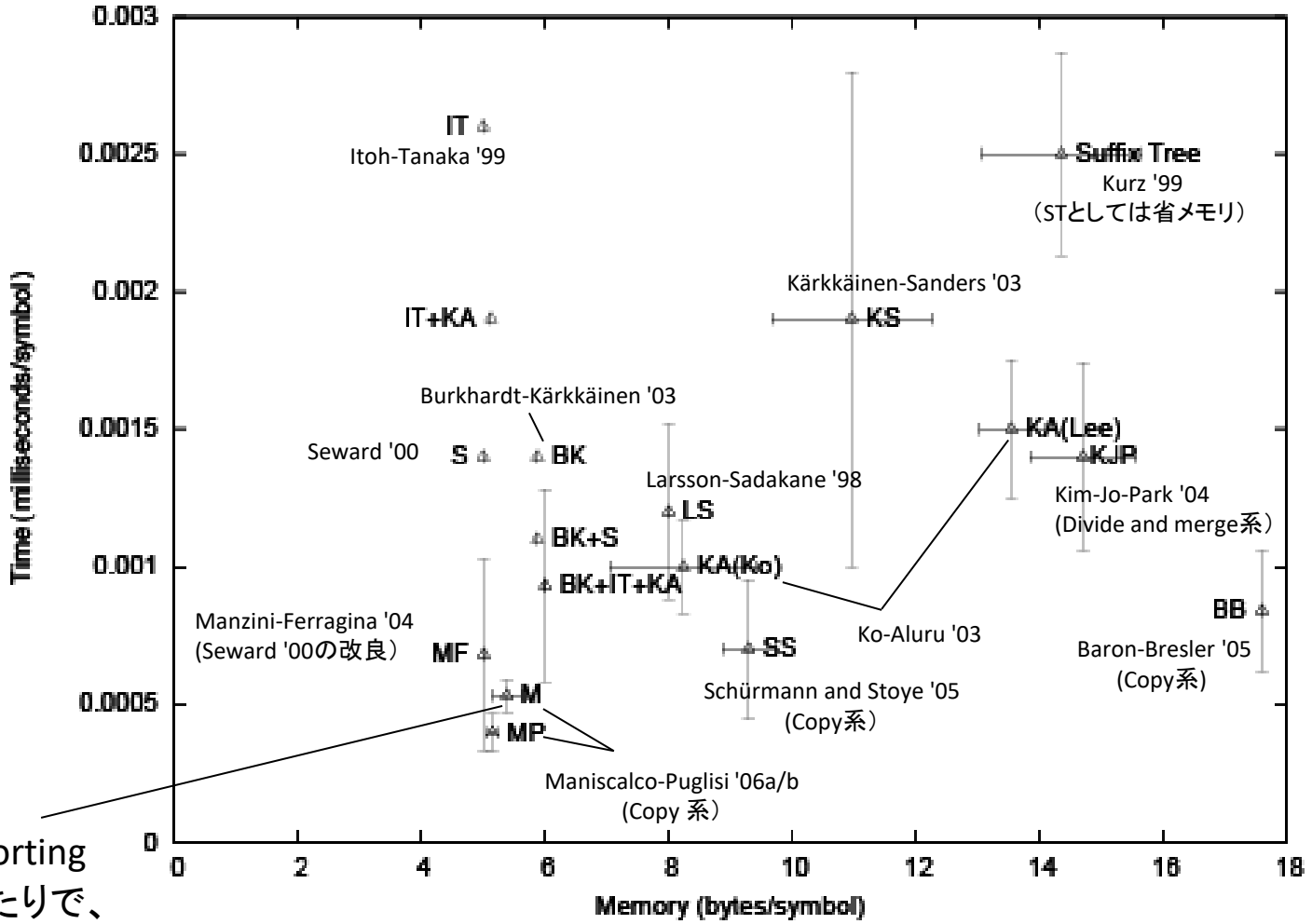


Induced Sorting (2)

- Ko-Aluruと同様 全接尾辞を「<」と「 >」の2つのタイプに分ける （線形時間）
- さらに連続する「<」の中で一番左のもの「<*」（LMS）を考える。
- Induced sorting ではこの 「<*」タイプのみをまずソートする（再帰）
  - LMSの数は全体の半分(正確には+0.5)以下
    - 「＜」と「＞」が交互に来る場合が一番多くなるが、その時の数
    - 実際にはもっと小さくなる
  - この部分問題サイズが他のアルゴリズムに比べてかなり小さいことが、全体のアルゴリズム性能の高い理由となっている
- 「>」タイプのソートは「<*」から「うまい」radix sortで可能
- 全「<」タイプのソートは「>」タイプから「うまい」radix sortで可能
  - ここが2段階となっている、というのがInduced sortingの新しいアイディア！
- それらのマージもKo-Aluruと同様に各比較は一文字目のみで可能（すなわち $O(1)$で可能！）



様々なアルゴリズムの計算機実験による実性能比較



Induced sorting はこのあたりで、最速、最小メモリ付近、でかつ理論的にも$O(n)$

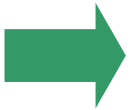
Fig. 8 from Simon J. Puglisi, W. F. Smyth & Andrew Turpin, "A taxonomy of suffix array construction algorithms", *ACM Computing Surveys,* 39-2 (2007).

接尾辞配列の最重要付加データ： 高さ配列(Height Array)

- Suffix Arrayにおいて、隣同士の接尾辞のLCP (longest common prefix) lengthの配列
  - ississippiとissippiのLCPは4
- 接尾辞配列から線形時間で計算可能 [Kasai et al. 2001]
- 様々な応用
  - RMQと組み合わせれば任意の接尾辞間でLCPが求められる
  - 検索は $O(m+\log n)$ にできる
  - 接尾辞木を線形時間で作成できる

Suffix Array

```
10: i$
 7: ippi$
 4: issippi$
 1: ississippi$
 0: mississippi$
 9: pi$
 8: ppi$
 6: sippi$
 3: sissippi$
 5: ssippi$
 2: ssissippi$
```

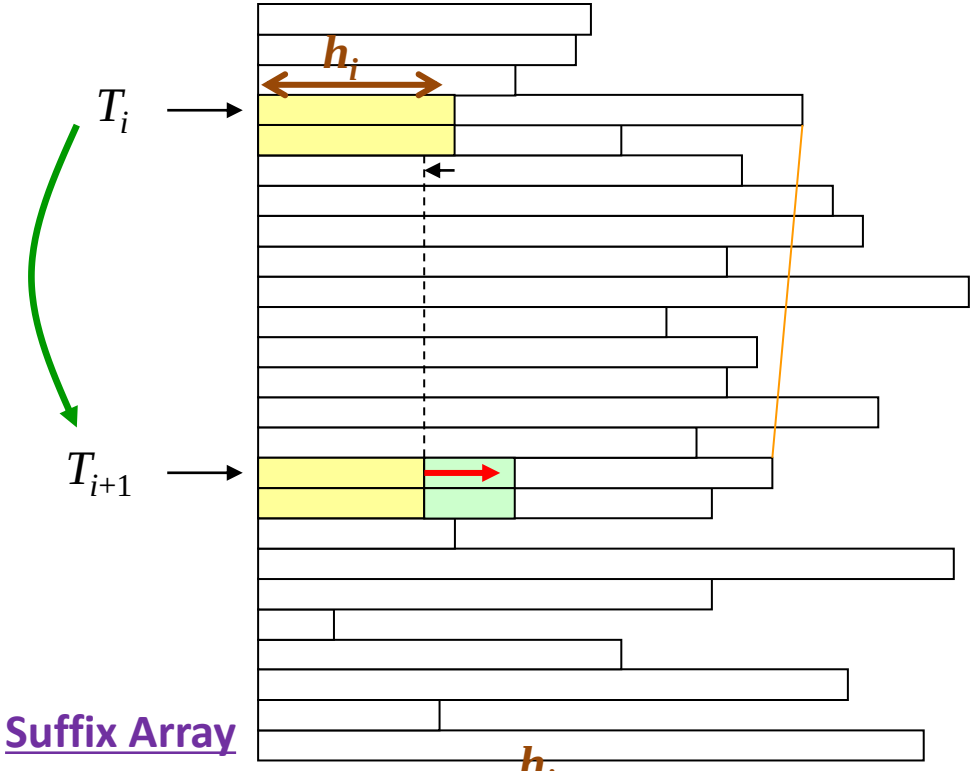


```
1
1
4
0
0
1
0
2
1
3
```

高さ配列

高さ配列の作成法

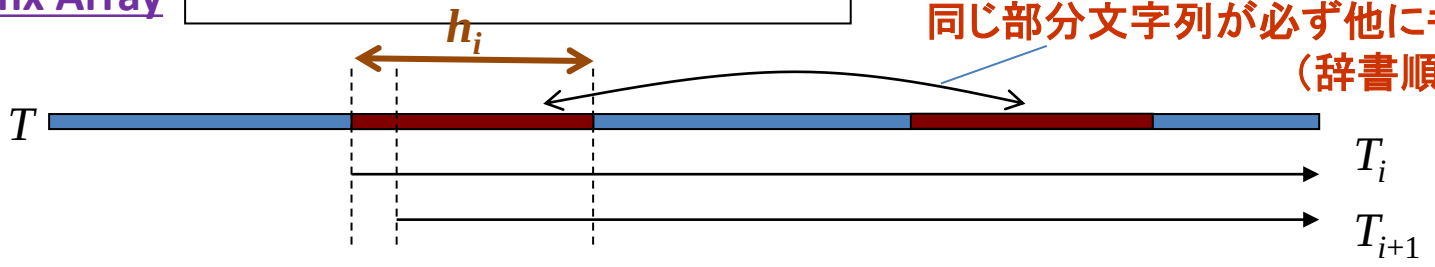
- 線形時間アルゴリズムが存在（しかも、アルファベットサイズに関係ない！）
  - 元の文字列の位置の順番で高さ配列を作成するだけ！
    - 逆接尾辞配列を用いる
  - ナイーブに作成すると$O(n^2)$必要



一つ前の接尾辞に対応する高さ配列の値より1小さい位置から調べる

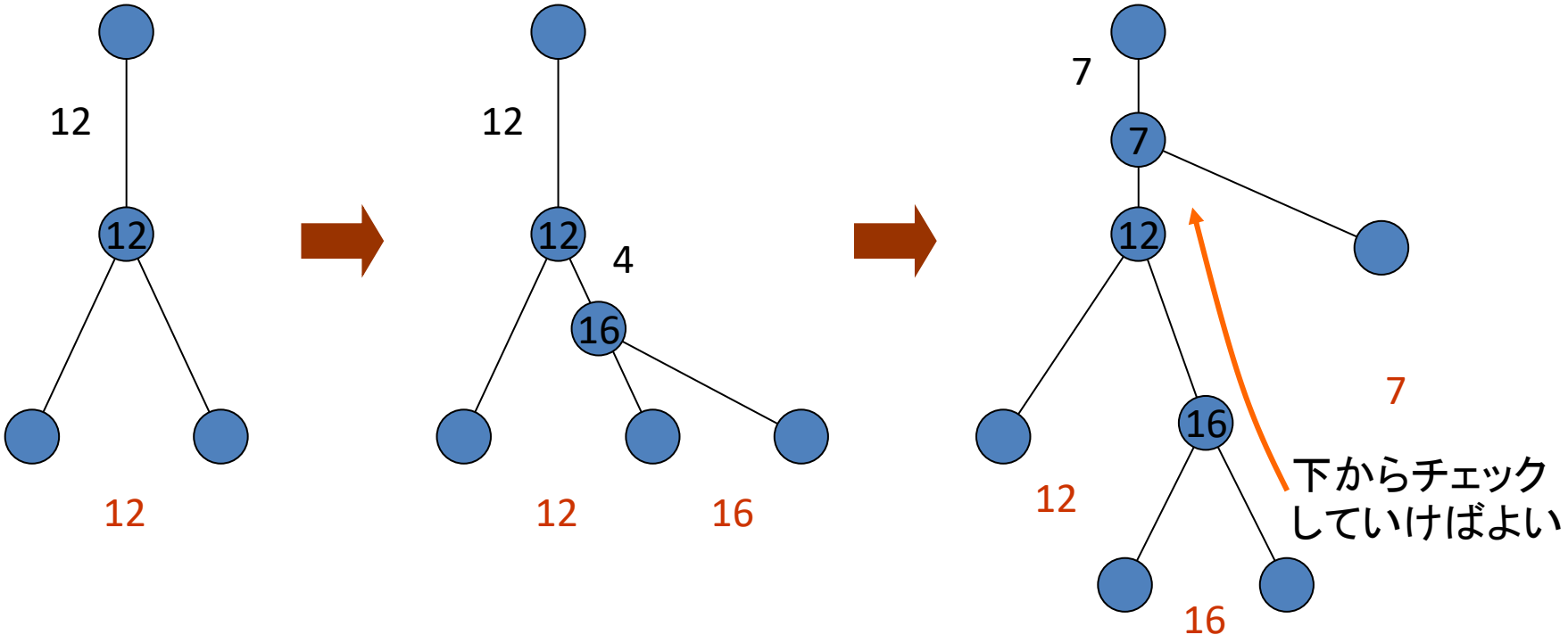
↓

どう頑張っても$2n$以上チェックすることはない！



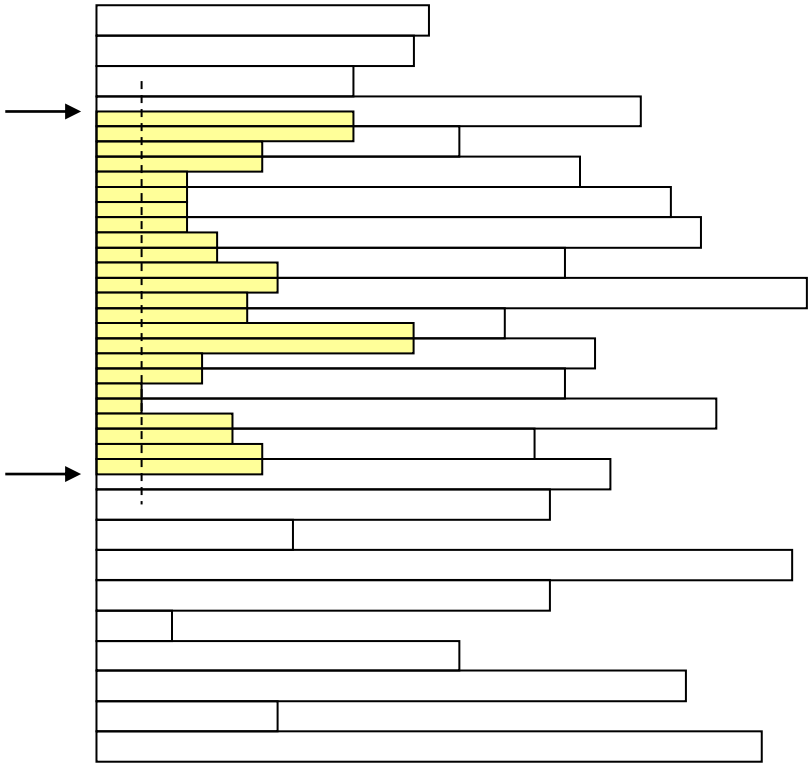
接尾辞配列→接尾辞木

- Height Array さえあれば、$O(n)$ で接尾辞木を構成可能
  - 左から順に作っていけばよいだけ
    - RMQの時のCartesian Treeの作成方法と似たアルゴリズム



LCPの計算

- LCP = Height Array + RMQ



Suffix Array

Height Array の該当する範囲内で最も最小値を求めるだけでよい

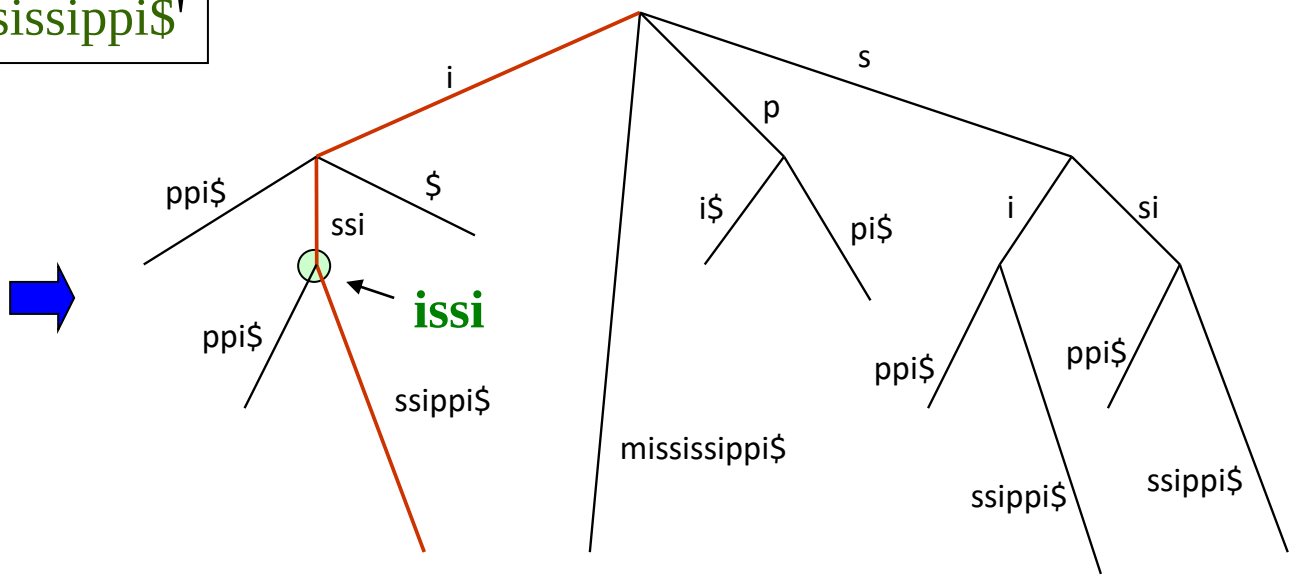
応用1： 接尾辞木の最も基本的な応用：Exact matching

- 作成して上から辿るだけ（接尾辞木）
- 二分探索、逆方向探索等で探索（接尾辞配列）

Suffix tree of 'mississippi$'

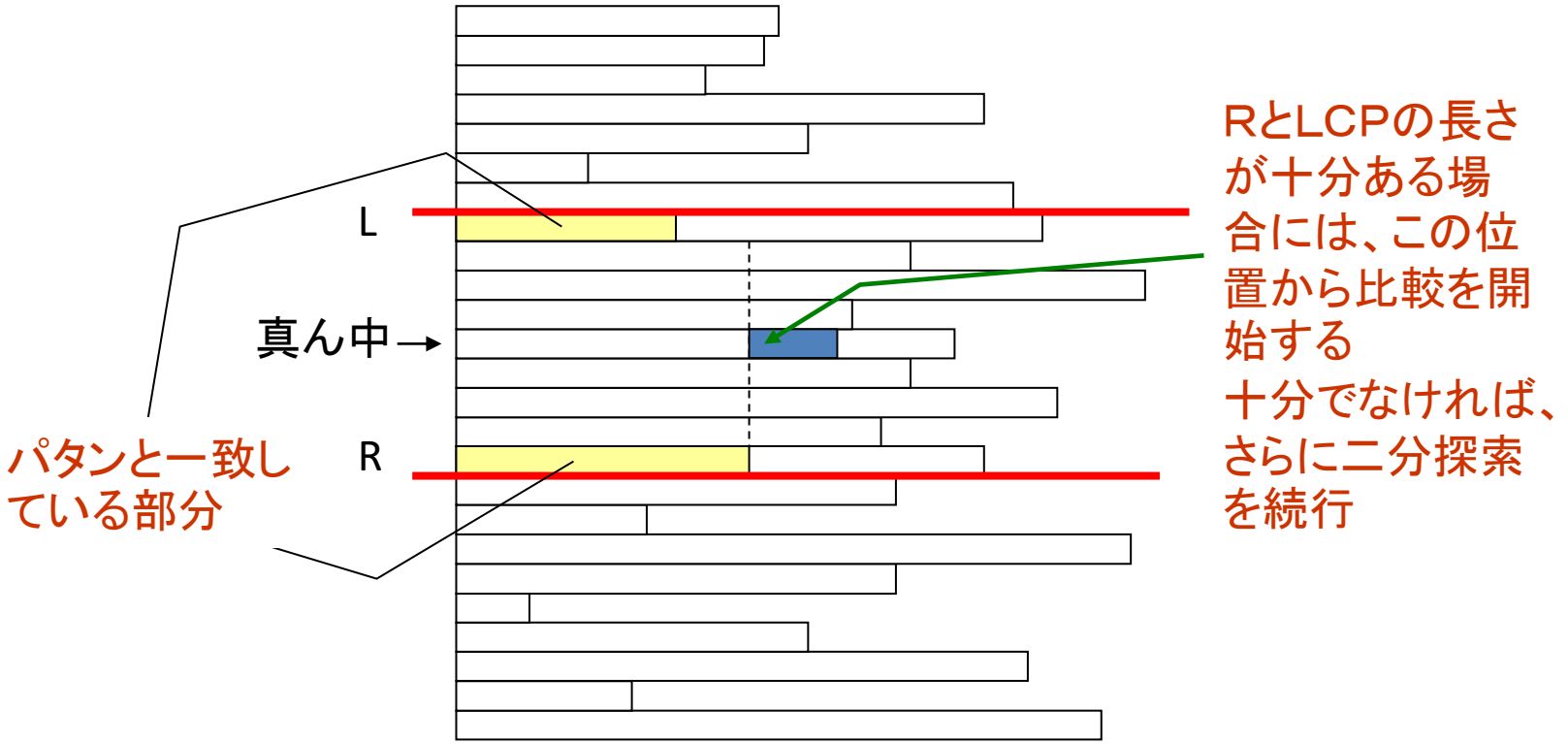
All the suffixes

mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$



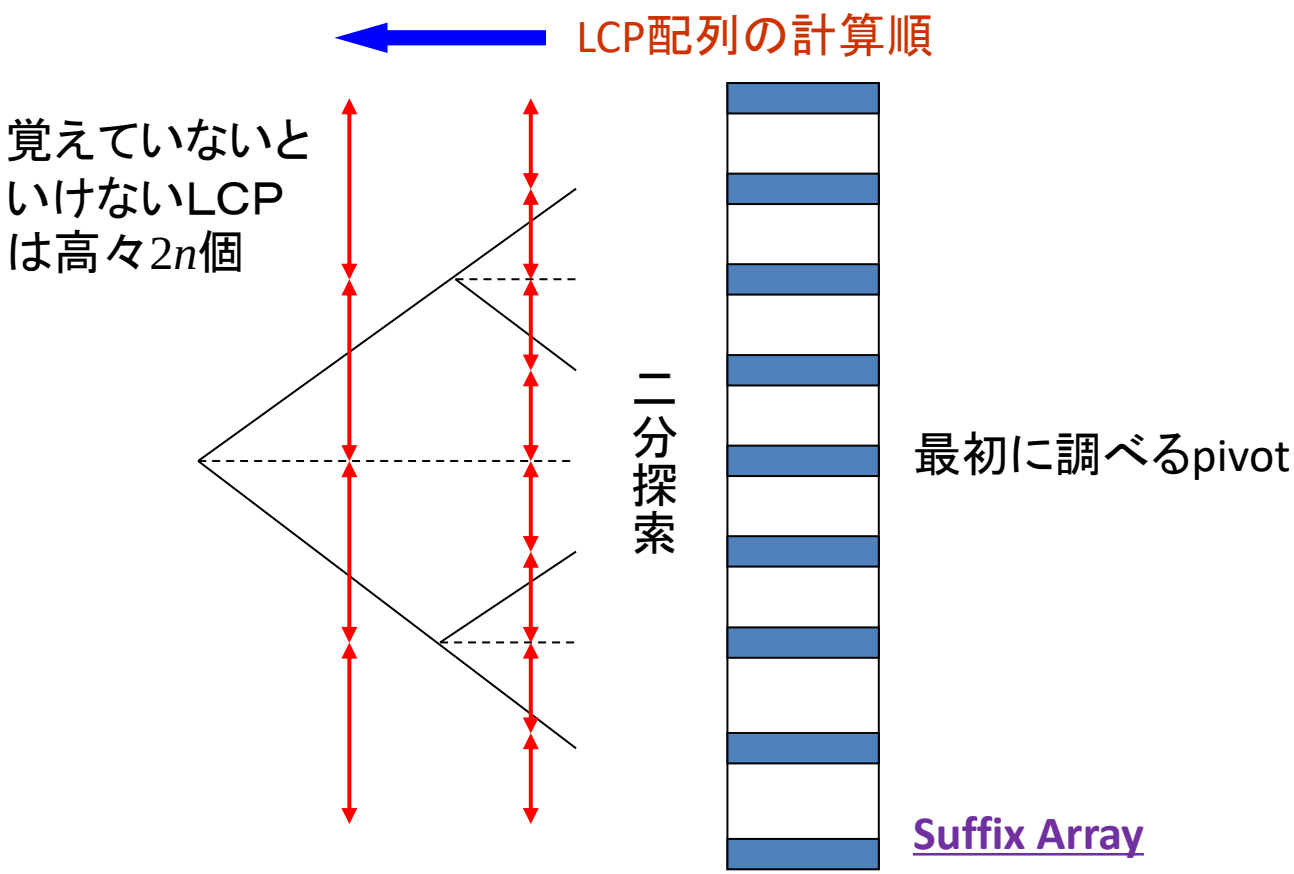
高さ配列を用いた検索

- RMQと組み合わせれば、LCPの長さが$O(1)$でわかるので、$O(m+\log n)$ で検索可能
  - 但し効率はよいとは言えない（実際には、最初に紹介した、「賢い検索」でも十分）



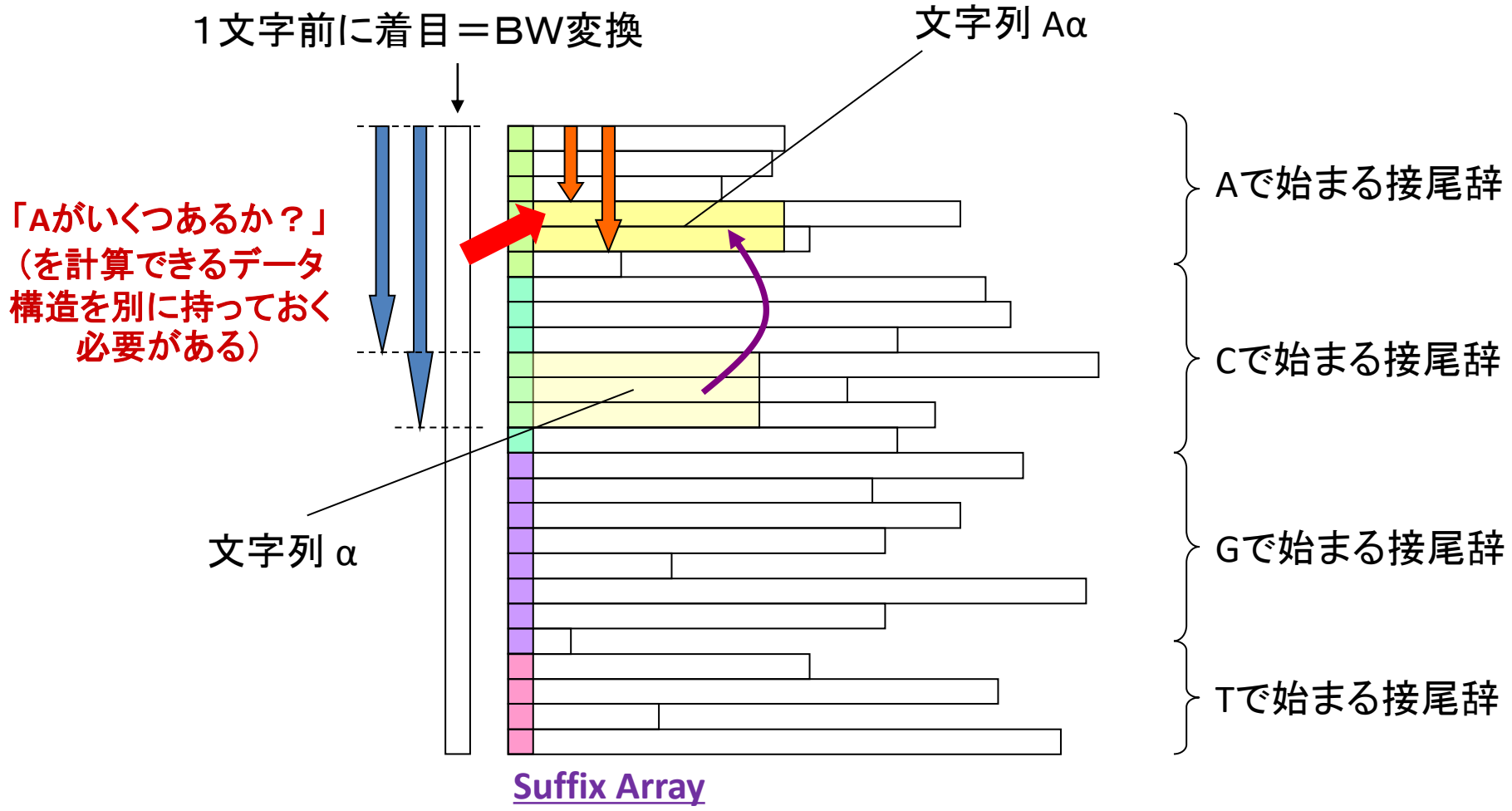
LCP Arrayによる検索 [Manber and Myers '92]

- $O(m + \log n)$ を直接可能にするデータ
  - 二分探索の枝分けは固定なので、必要なLCP長は$O(n)$個
    - 作成時間は$O(n)$ （高さ配列から計算）
    - MMのオリジナルの論文では $O(n \log n)$
      - 当時は高さ配列を $O(n)$ で作成する方法が知られていなかった



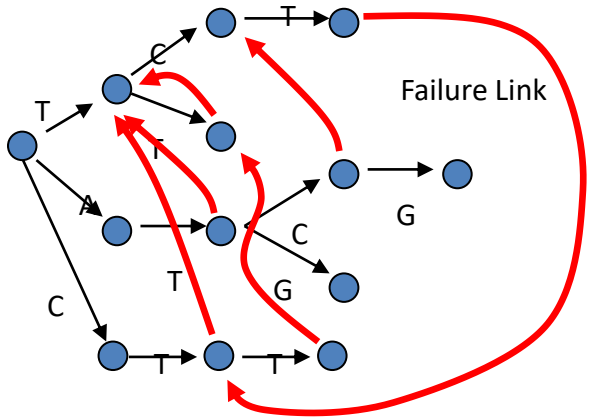
逆方向検索 （→ FM-Indexにつながっていく）

- なんと検索は逆方向にもできる！
  - Weiner Algorithmの逆suffix linkを辿ることに相当する
    - *cf.* BW変換（→圧縮アルゴリズムの紹介の中で紹介予定）



応用2： Set Matching Problem

- Aho-Corasick algorithm（復習）
  - Knuth-Morris-Pratt のマルチパタン化
  - $O(n+m+k)$ (n: テキストサイズ、m: パタンサイズの和、k: 出現数)
- 接尾辞木を用いたSet matching
  - 接尾辞木では普通に検索するだけでも$O(n+m+k)$が実現可能
    - テキストとパタンのどちらを前処理するか、の違い
      - どちらが変化するのかによってACかSTのどちらを使うかを決める
    - パタンセットのサイズ＞テキストサイズならばSTの方が省メモリ
  - **Matching Statistics**
    - これもパタン側の接尾辞木を用いる
    - これでも$O(n+m+k)$が可能
    - 他の応用も多い
  - Boyer-Mooreをマルチプル・パタン化
    - パタン側の接尾辞木を用いる

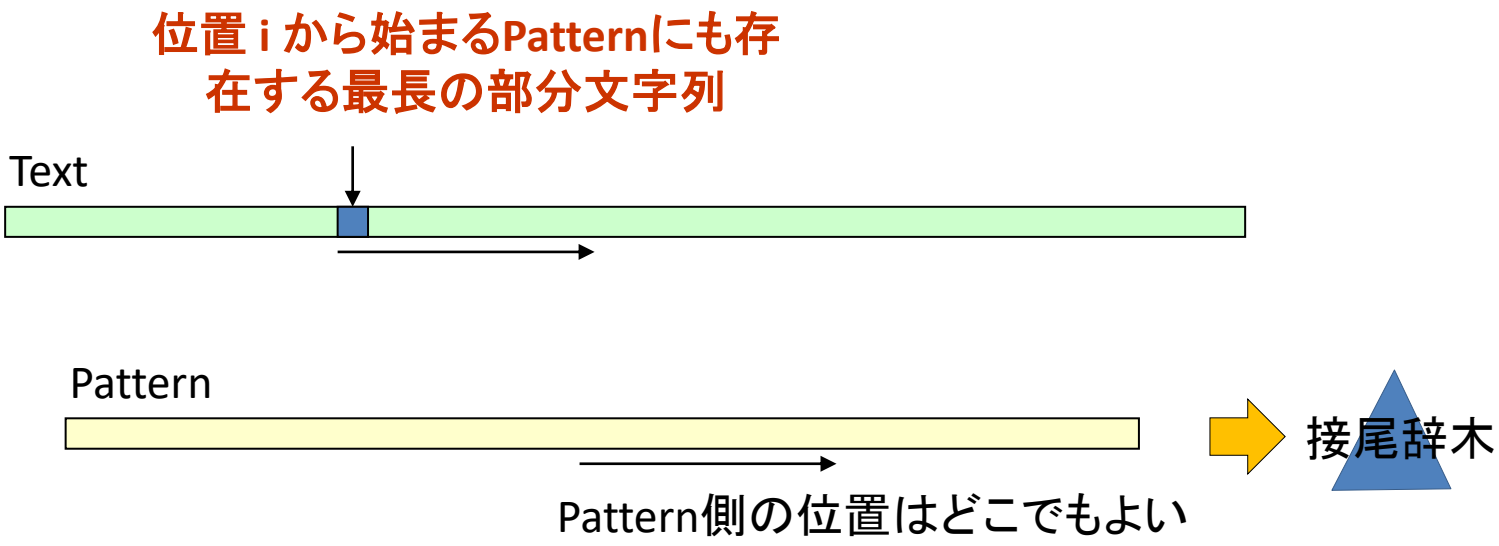


Aho-Corasick Automaton

Matching Statistics (1)

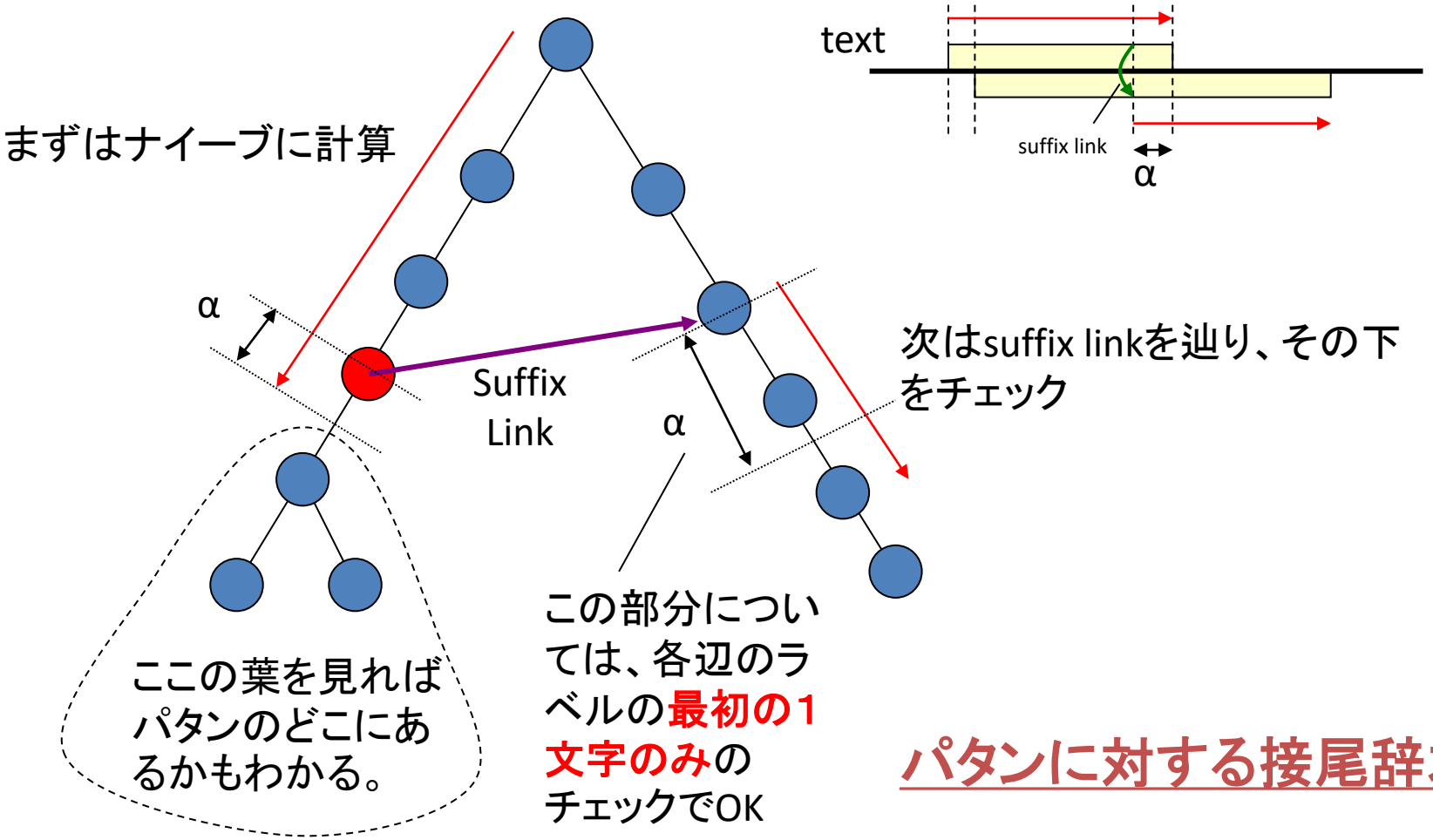
Matching Statistics とは

- テキスト文字列のすべての位置に関して、その位置から始まる最長のパタン文字列との共通の部分文字列長を計算したもの
- パタン文字列に対する接尾辞木を用いれば線形時間で可能



Matching Statistics (2)

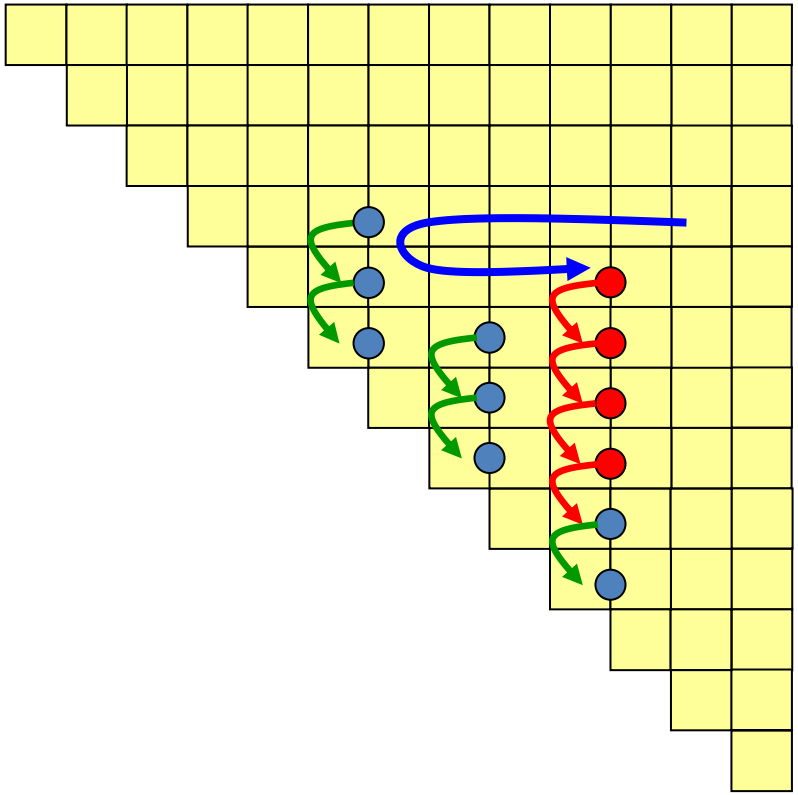
- パタン側の接尾辞木を用いる
  - Suffix Linkを用いる



パタンに対する接尾辞木

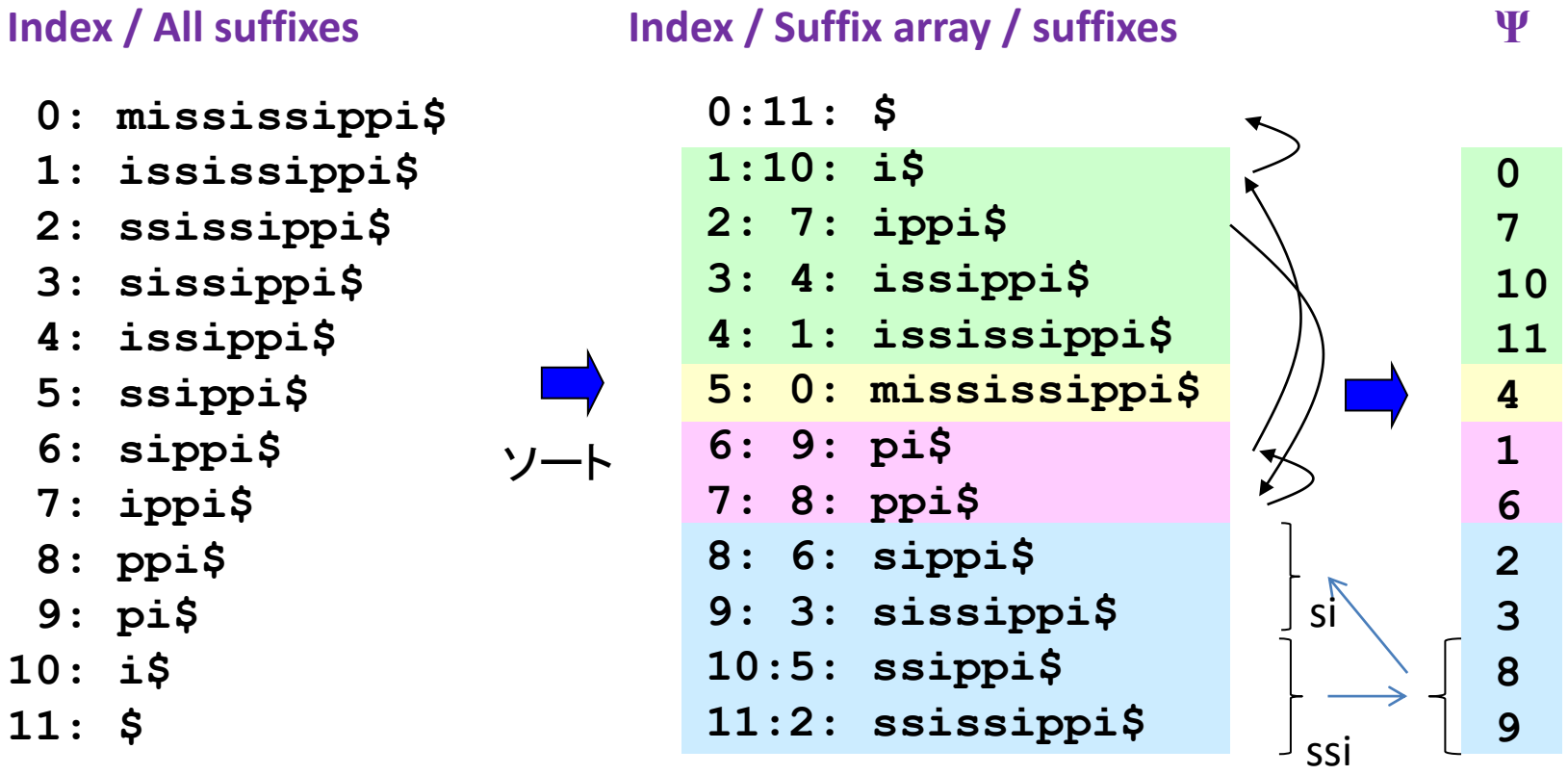
Matching Statistics (3)

- 計算時間は $O(n)$
  - 辿るノード数が最大でも n であるため
    - Ukkonenのアルゴリズムが$O(n)$である理由と同じ



Suffix ArrayでMatching Statisticsはできるか？

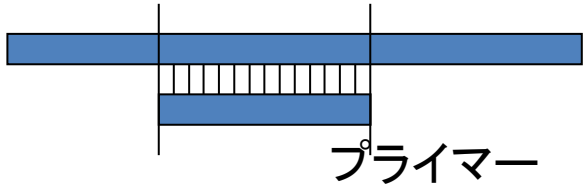
- Ψ配列 (接尾辞配列上のSuffix Link)を用いればある程度可能
  - プサイ
  - 逆接尾辞配列から線形時間で作成可能
    - Ψ配列は接尾辞配列を圧縮するのにも用いられることがある
  - 二分探索は必要→ $O(n \log n)$
    - そのオーバーヘッドをなくすには、さらなる付加データ構造が必要



たとえばssi　(SA[10..11])のsuffix link先のsiはSA[8..9]

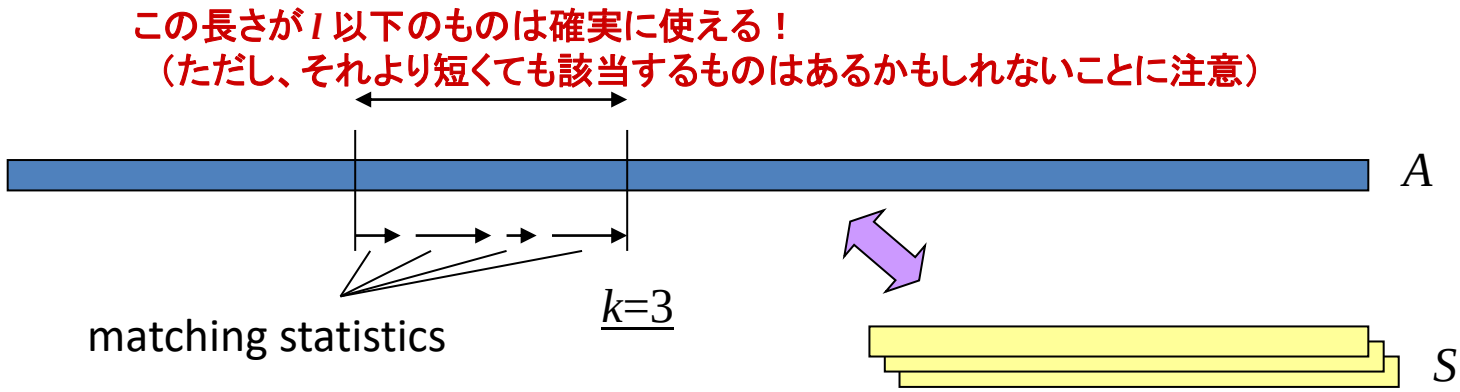
Matching Statisticsの応用1：プライマー設計 （1）

- プライマー
  - あるDNA試料に対して、ある短い配列が存在するかどうかを実験的に検査する際の、その短いDNA配列のこと
- プライマー決定問題
  - 入力
    - 配列 A と配列集合 S
      - Aを他(S)の配列と実験的に区別できる部分配列を求めたい
  - 出力
    - 配列 A のすべての位置iについて、その位置から始まる部分文字列で S に存在しない一番短い文字列 （＝ AにはマッチするがSの配列にはマッチしない）
  - 意味
    - その場所から始まる断片（あるいは遺伝子）が資料の中に含まれるかどうかを調べたい
  - アルゴリズム
    - Matching statisticsで求めた値 +1



Matching Statisticsの応用1：プライマー設計 （2）

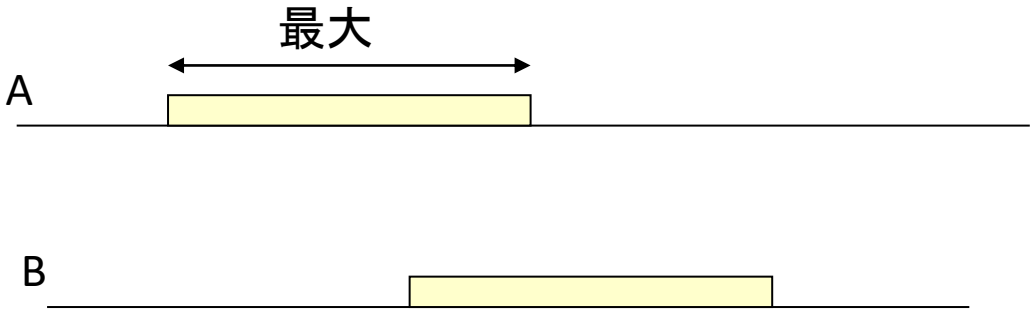
- 問題を少し難しくして、次の問題を考える
  - S のどの部分文字列ともedit distanceが k 以上である最小の部分文字列（ただし長さ l 以下）を、Aのすべての位置について求める



- ただし実際には、更に多くの条件がある
  - プライマーとして使えるためには、複雑な条件がある
    - GC含量が50%前後、回文禁止
  - プライマーは2つ一組で用いられることが多い
    - 組み合わせを考える必要
    - 2つで囲んだ領域の長さが異なるものも判別可能 （電気泳動）

Matching Statisticsの応用2：Longest Common Substring

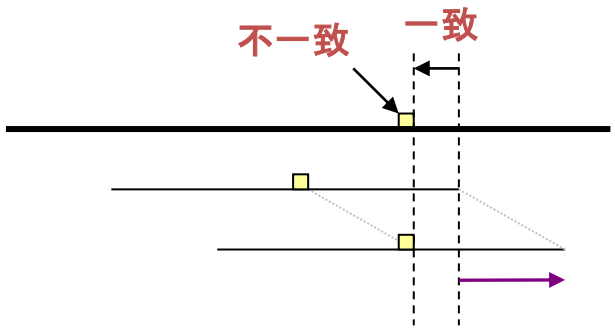
- Matching Statistics で最大値をとるものを探す
  - 当然だが線形時間で見つけられる

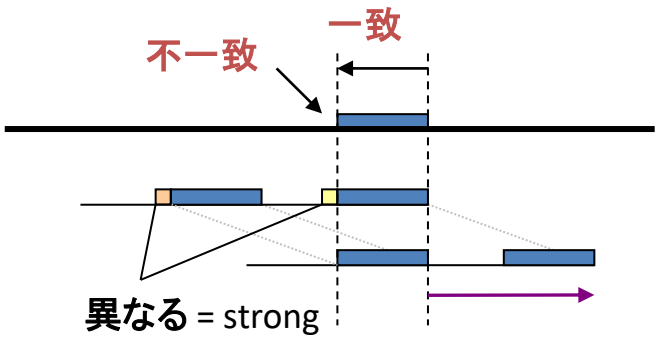


注：ただしMatching Statisticsを使わなくてもよい→次の話題

（おまけ） Multi-pattern Boyer-Moore というのもある　(1)

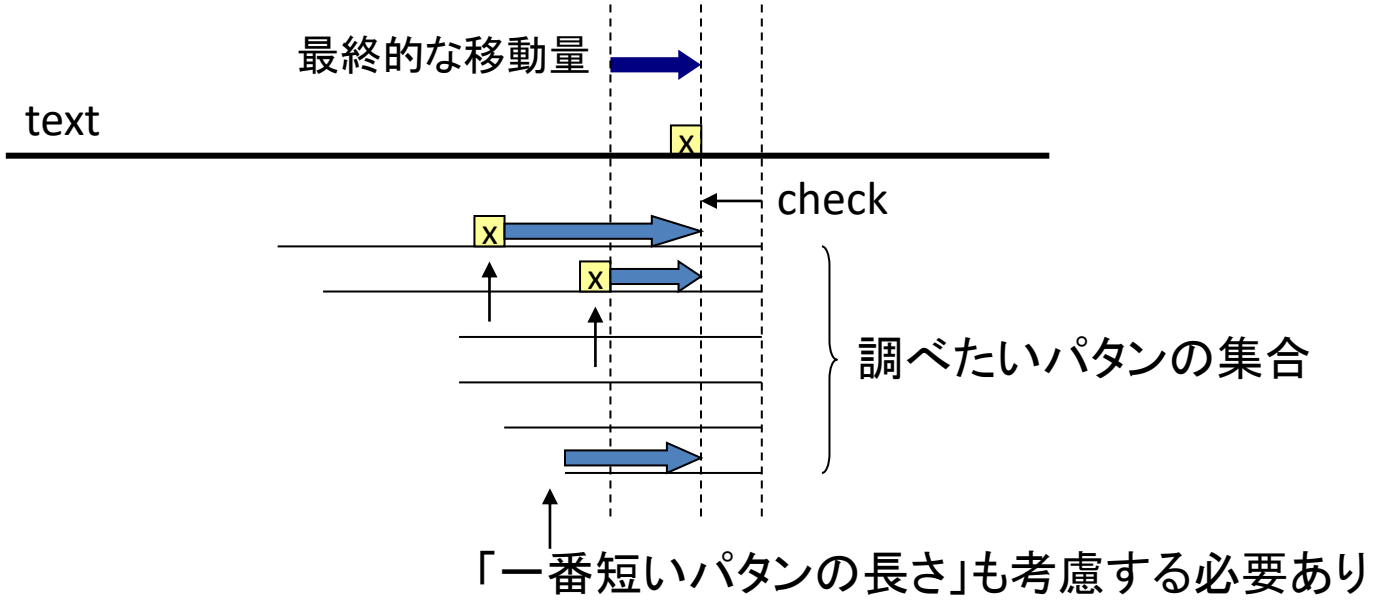
- Boyer–Moore（復習）
  - 不一致ルール (bad character rule)
    - チェックした文字がxであれば、パタンの中のxという文字のうち最後の位置までずらせる
  - 接尾辞一致ルール (strong good suffix rule)
    - 後ろk文字が一致した場合、パタン中のそれ以外の場所にそのk文字が存在する(いくつかある場合は最後のもの)時、そこを一致させるようにずらす
  - KMPより速いことが多い
- これをマルチプル・パタン化することを考える





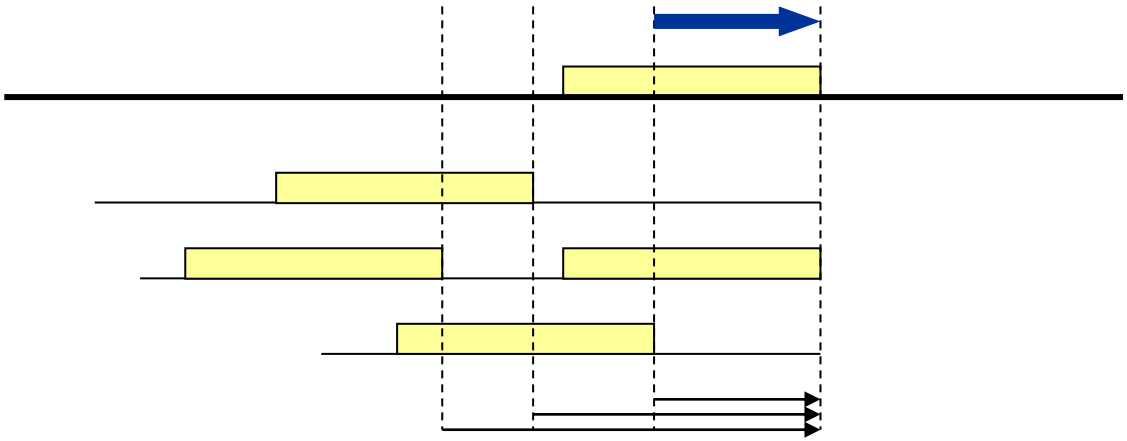
Multi-pattern Boyer-Moore (2)

- 不一致ルールの拡張 （簡単）
  - 一番後ろにある不一致文字に一致する文字に着目
  - 一番短いパタンの長さに注意



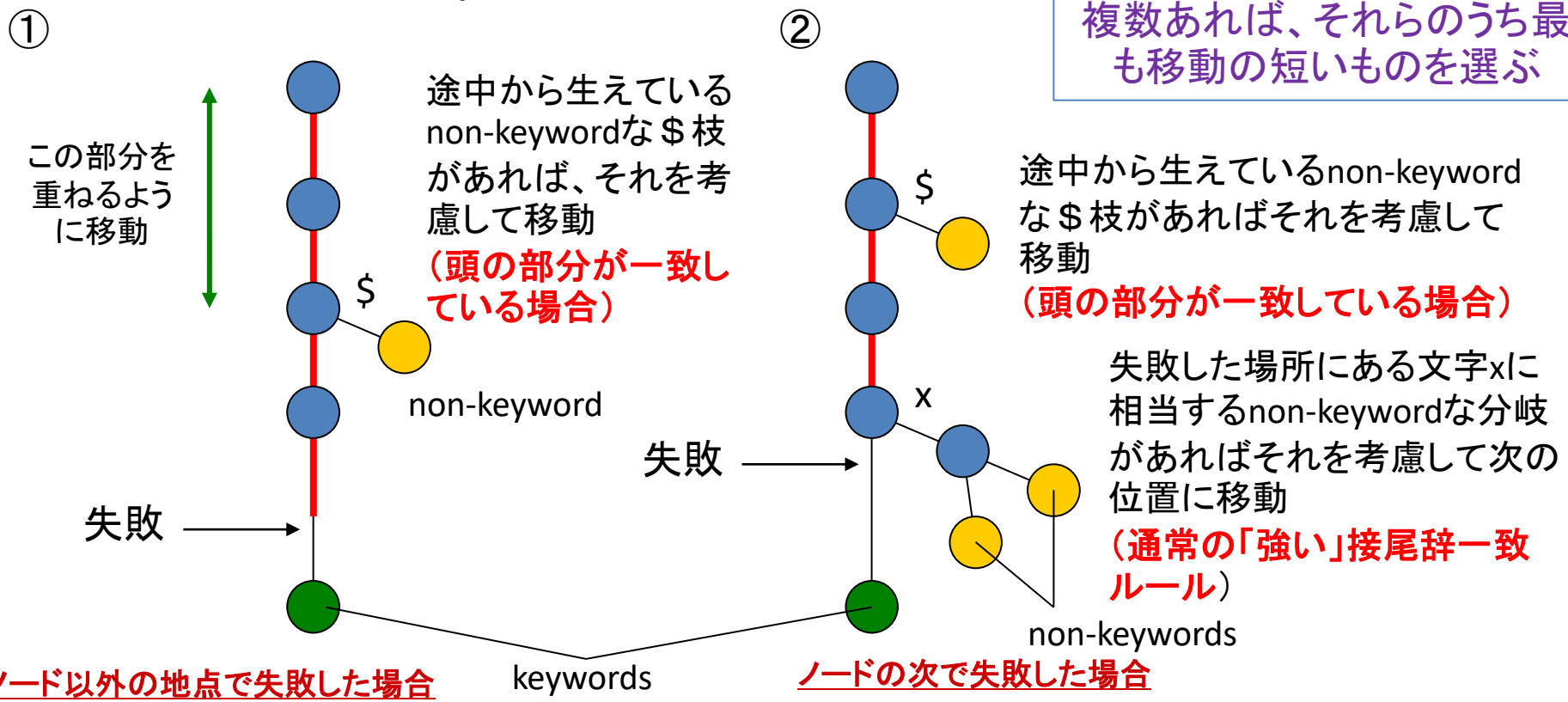
Multi-pattern Boyer-Moore (3)

- 接尾辞一致ルール　（これを接尾辞木を使って計算）
  - 逆順でのパタン集合のキーワード木を作成
    - 後ろからパタンのマッチングを検査するため
    - この木は↓の一般化接尾辞木に含まれる
  - 各ノードでの移動量を計算
    - 逆順でのパタン集合の一般化接尾辞木を利用
      - 一般化接尾辞木(generalized suffix tree)：複数の文字列に対する接尾辞木



Multi-pattern Boyer-Moore (4)

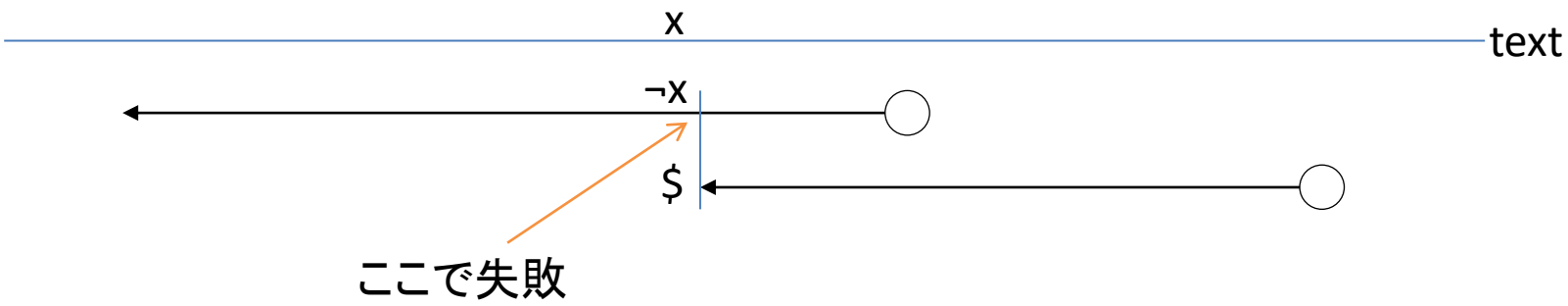
- 接尾辞木において
  - 各パタンに相当する枝とそれ以外を区別
    - 相当する枝＝keyword tree
  - それぞれのノードまで探索した時の移動量を計算
    - 探索はkeywordに相当する枝のみ



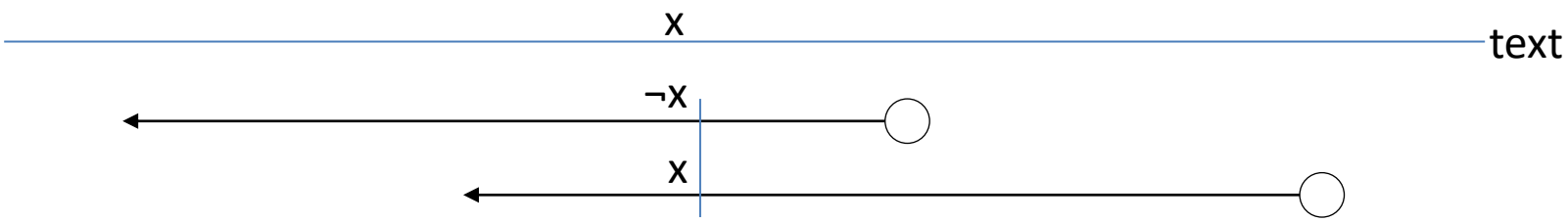
複数あれば、それらのうち最も移動の短いものを選ぶ

Multi-pattern Boyer-Moore (4)

①

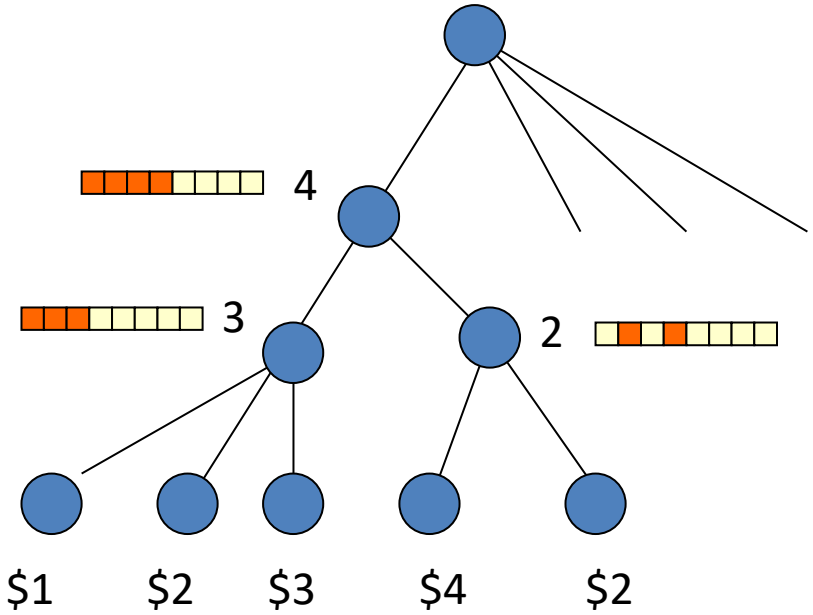


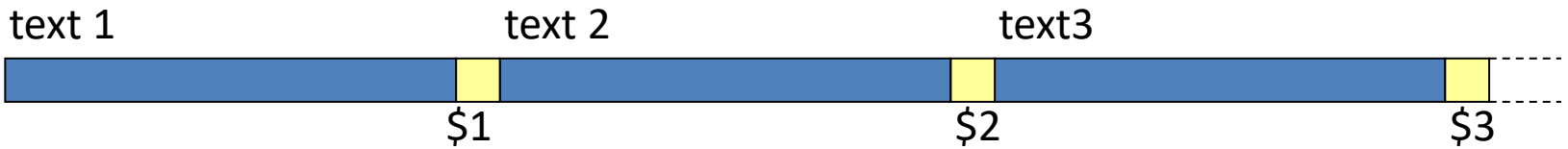
②



応用3： Multiple Common Substrings (1)

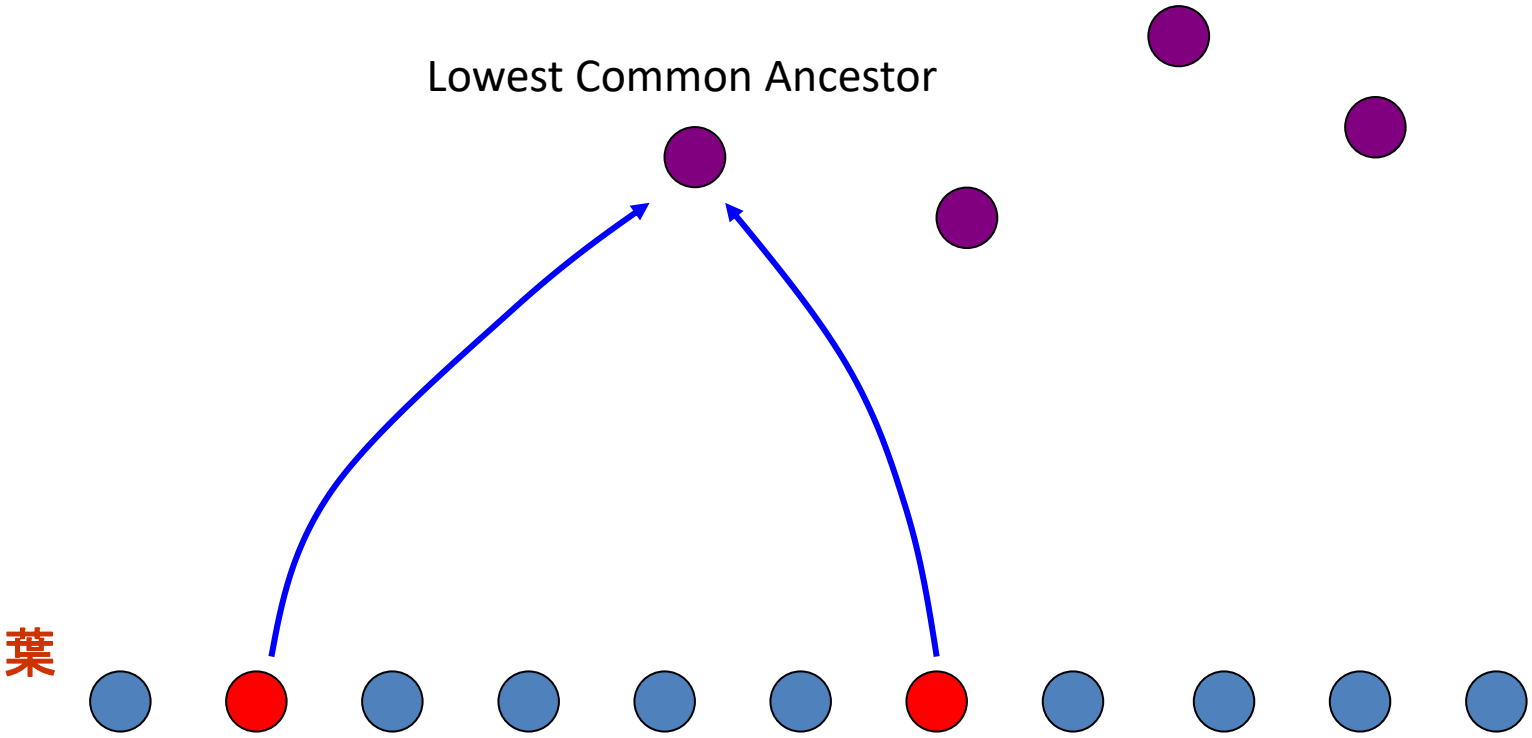
- 複数の文字列の間で共通に持つ文字列を求める問題
  - Longest ～はその中から最長のものを選ぶ問題
- Generalized Suffix Treeを作る
  - あるノードがいくつの文字列を葉として持つかを計算する
    - ナイーブにやると$O(kn)$（k: #patterns, n: sum of lengths）





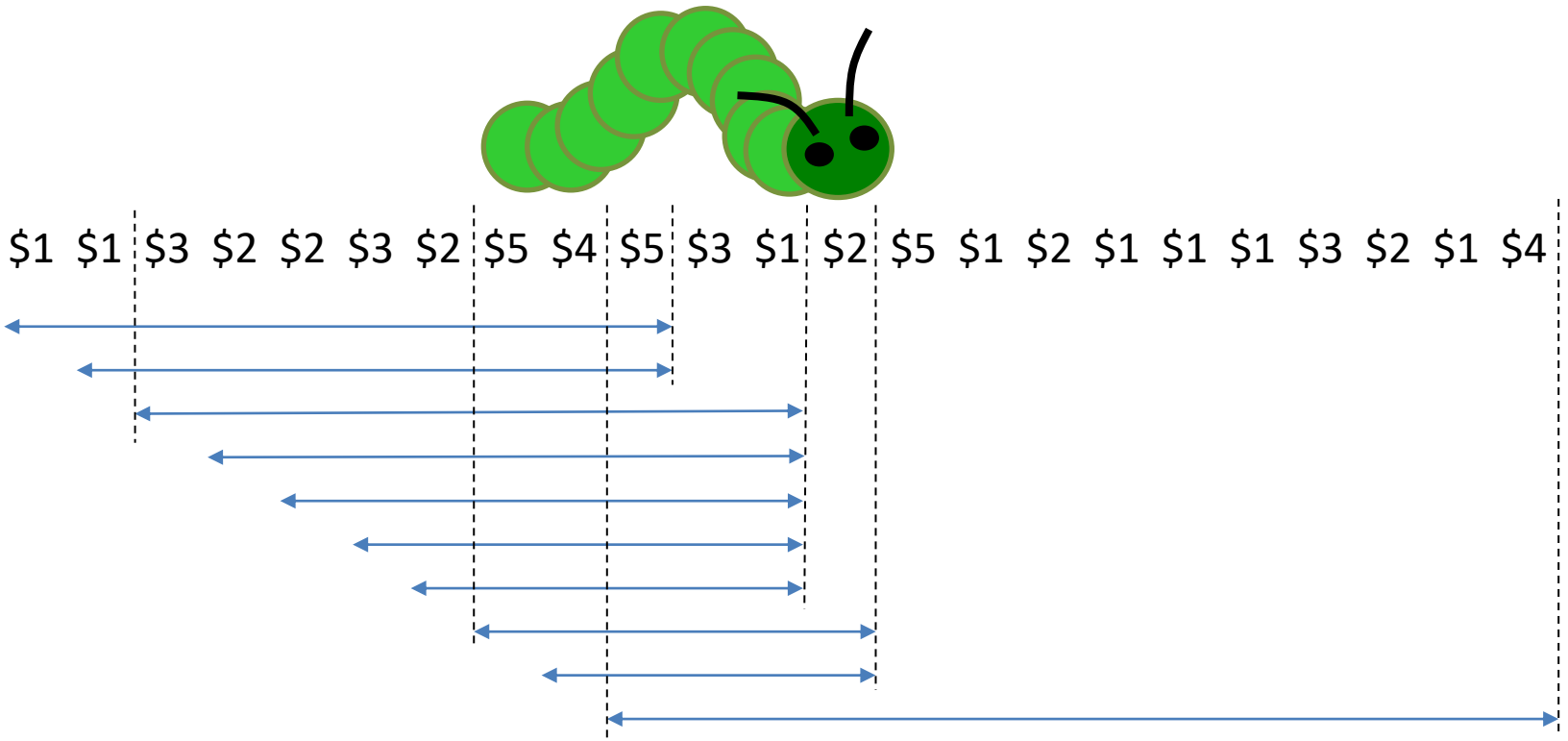
Multiple Common Substrings (2)

- LCAを用いれば線形時間にできる！



Multiple Common Substrings (3)

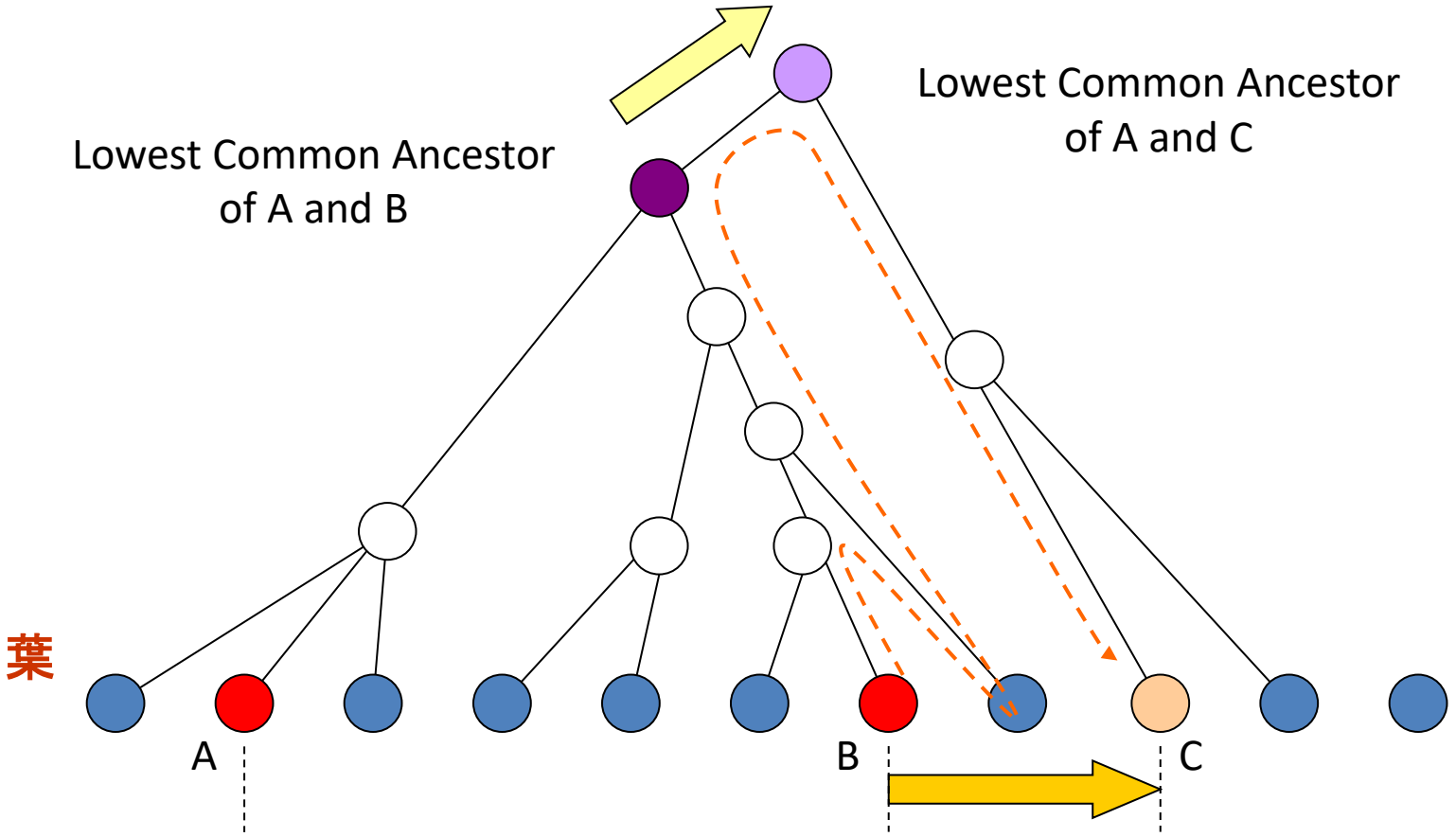
尺取り虫の例



$1～$5が存在する5本の文字列の場合例

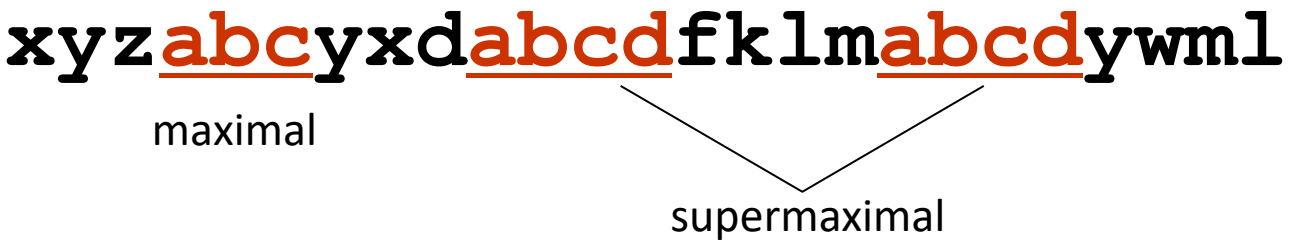
Multiple Common Substrings (4)

- 必要なLCAの計算
  - 実は単にTraverse するだけで線形時間で計算可能
  - よって複雑な $O(1)$ のLCA Query Algorithmは不要



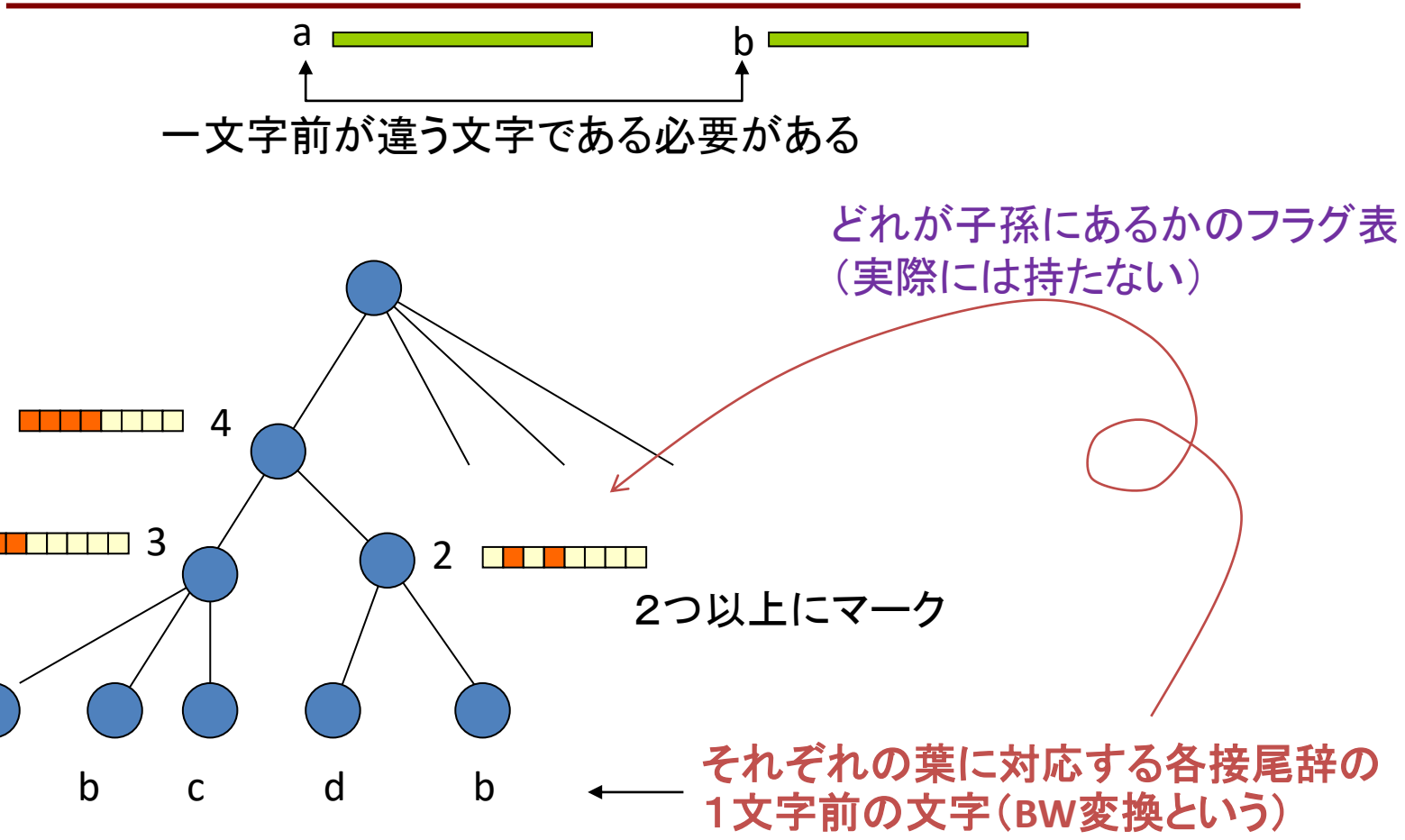
応用4： Maximal Repeats & Supermaximal Repeats

- Maximal repeat
  - identicalな二つ以上の部分文字列の組で、左右どちらでも1文字増やしたらrepeat数が変わるもの
- Supermaximal repeat
  - 他のmaximal な部分文字列の部分文字列にはなっていないようなmaximal repeat



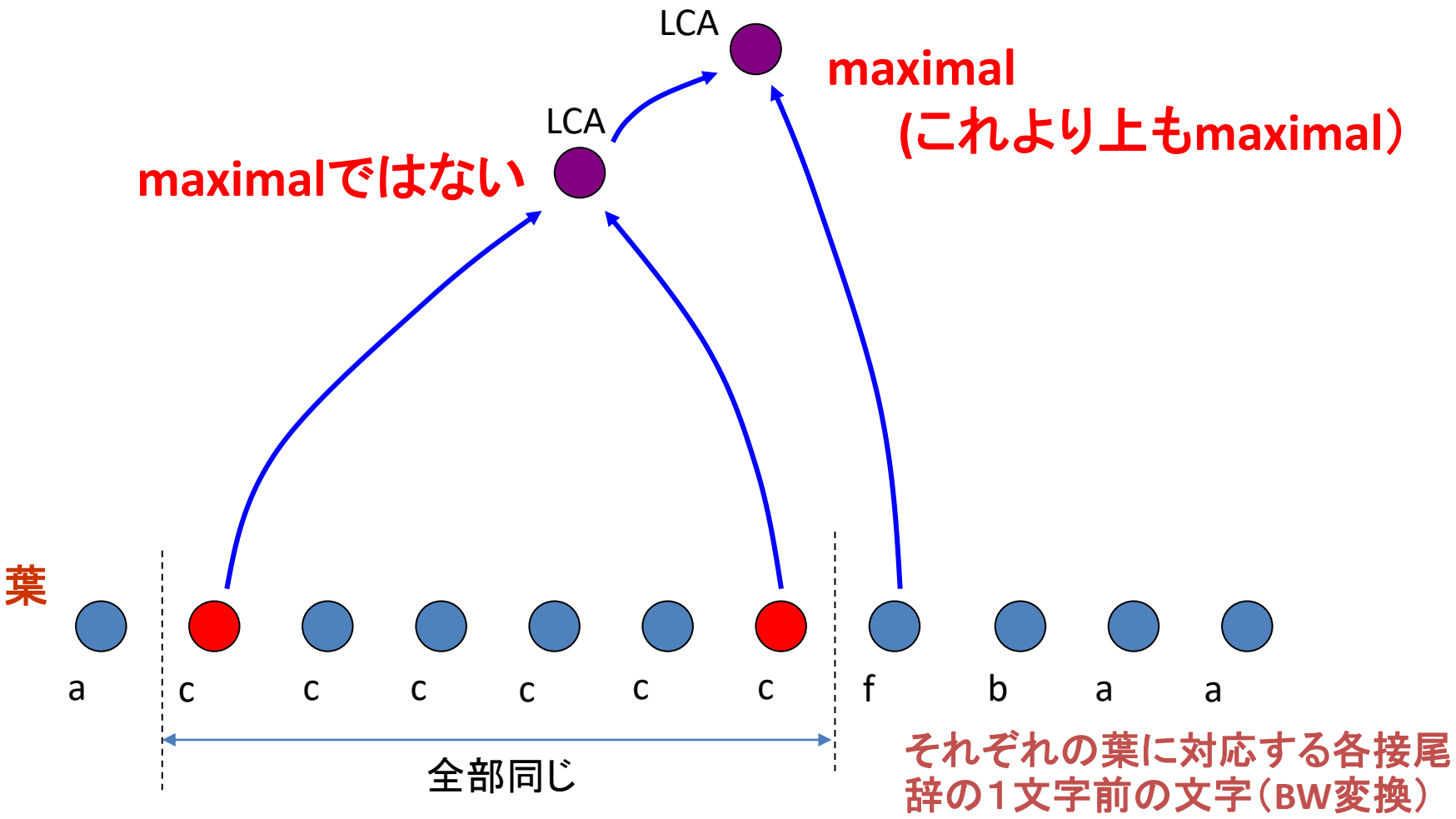
Maximal Repeats (1)

- 右側条件は「内部ノード」であること、でOK
- 左側条件はMultiple Common Substringsと状況が似ている
  - が実はもっと簡単（次ページ）



Maximal Repeats (2)

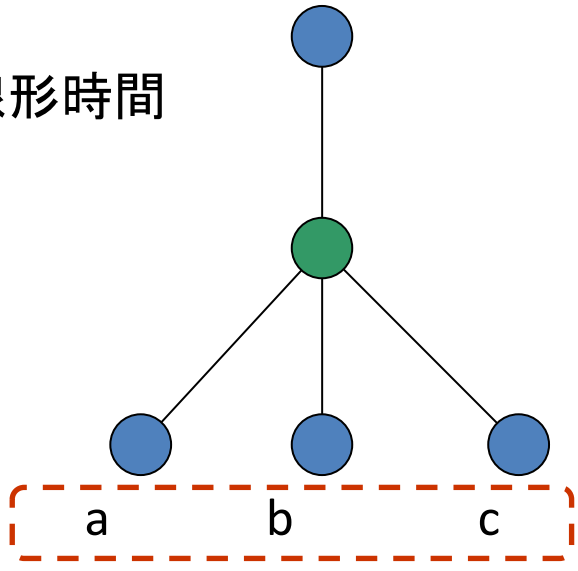
- 1種類しかない区間を除外するだけでよい→線形時間！



Supermaximal repeat

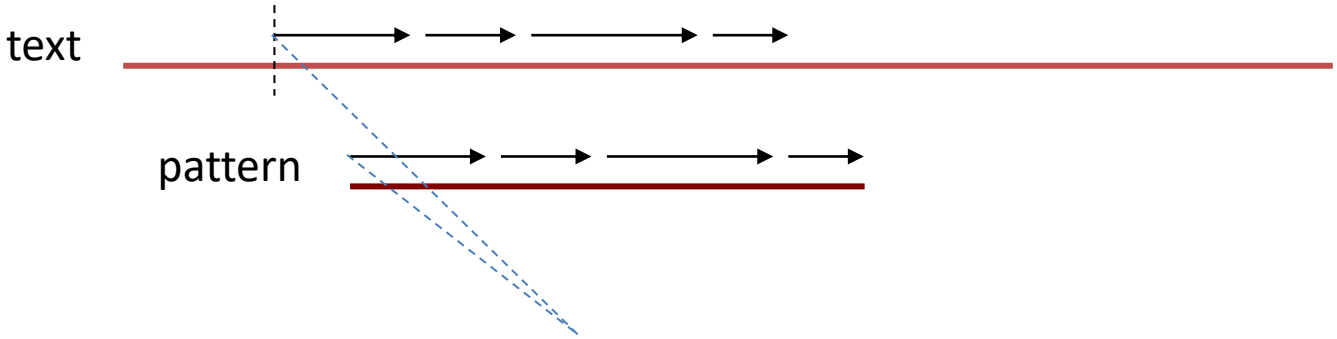
- 条件
  - 右側条件：子供はすべて葉である
  - 左側条件：子供に対応する接尾辞の一文字前の文字がすべて異なる
- 計算時間
  - アルファベットサイズが固定の場合線形時間
  - アルファベットサイズが固定でない場合
    - $O(n \log s)$　（s: アルファベットサイズ）
    - ハッシュを用いれば、計算時間の期待値は線形時間

直前の文字はすべて異なる必要



応用5：（LCAの応用2） k-Mismatch Problem

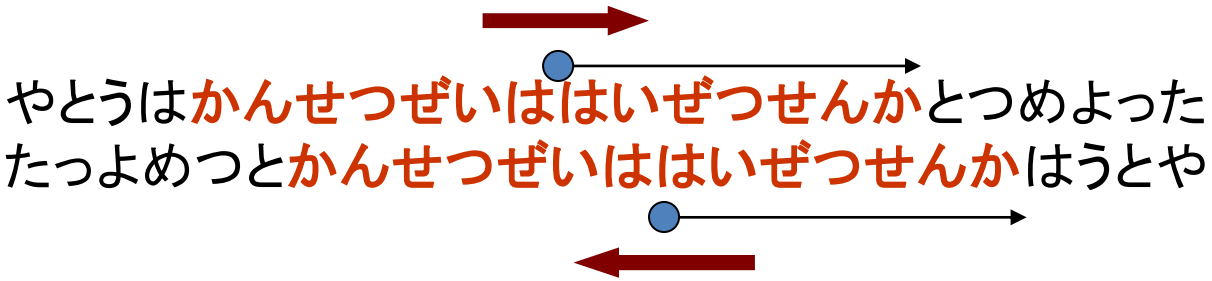
- ギャップを考えないミスマッチを k 個許すマッチング
  - $O(kn+m)$（n: テキスト、 m: パタン）
  - 一般化接尾辞木を作成し、LCPを用いる



ここから始まる共通な最長部分文字列の
長さをLCPで一発で求めることができる

応用6：（LCAの応用3） 回文(palindrome)をみつけるには

- 逆順の文字列との一般化接尾辞木を作り、各位置に対応するLCPの長さを求めればよい
  - 線形時間で可能
  - DNA,RNAでの相補配列を考える場合でもやることは同じ
    - ATCCTGCAGGAT など （これも生物学的にpalindromeと呼ぶ）
    - 間に最大 i 文字の隙間をいれるのを許す場合は$O(i \cdot n)$
  - k−mismatchを許す場合は$O(k \cdot n)$



応用7：（LCAの応用4）Tandem Repeat (1)

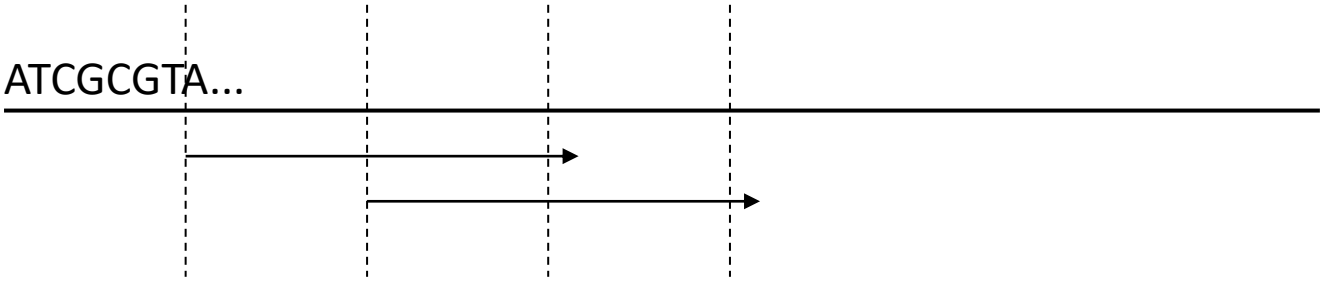
- タンデムリピート
  - 同じ（類似した）文字列の連続した繰り返し
  - 様々な関連現象
    - 類似遺伝子の生成
      - 視覚における赤と緑を知覚する遺伝子は互いに類似、連続して並んでいる。（進化の過程でコピーされて分化した）
    - 個人の特定
      - 基本単位の短いリピートの繰り返しの数が個人によって異なることがあり、しかも検出が簡単　→　犯罪捜査
    - がん細胞DNAの解析
      - Copy number variation (CNV)
    - DNA修飾・エンハンサー等への影響？
    - 寿命を決定？
      - テロメア　（DNA端にある、DNAを保護するため（？）の繰り返し）
    - その他さまざまな機能既知・未知の繰り返しが存在

Tandem Repeat (2)　超ナイーブ法

- すべての位置i ($1 \le i \le n-1$)とすべての長さ j ($1 \le i \le (n-i)/2$)に対して
  - 位置iから始まる長さjの部分列を繰り返し単位とする繰り返し区間があるか？　あればその長さはどれくらいか？

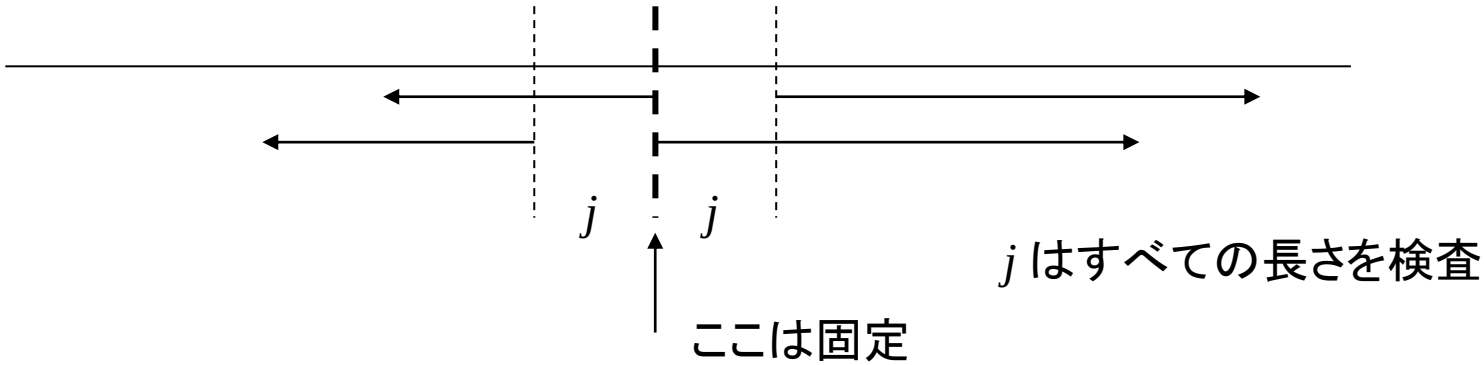
を計算する

- 計算時間は一つのi, jの組に対し$O(n)$の計算時間がかかるため、全体では$O(n^3)$となる
- ただし、入力がランダムな場合、一つのi, jの組に対する計算時間の期待値は $1+1/2+1/4+1/8+....\rightarrow 2$ 回の比較ですむので、$O(1)$である。よって平均計算量は $O(n^2)$



Tandem Repeat (3)　分割統治法

- 繰り返し区間を3つに分類する（ $m = \lceil n/2 \rceil$ ）
  - A)　区間が$W[m]$を含むもの
  - B)　区間が$W[1..m\text{-}1]$に完全に含まれるもの
  - C)　区間が$W[m\text{+}1..n]$に完全に含まれるもの
- A)に関しては、$W[m]$から左右に繰り返し列があるかをチェックする
  - ナイーブに計算して、$O(n^2)$
  - 入力がランダムの場合の平均は $O(n)$
- B)及びC)に関しては再帰的に計算する
  - すでに見つかっている繰り返し区間よりも短くなると終了
- 全体の計算時間は $O(n^2)$
  - この場合、$O(n^2 \log n)$とならないことに注意　←漸化式から計算
  - 入力がランダムの場合の平均計算時間は　$O(n \log n)$



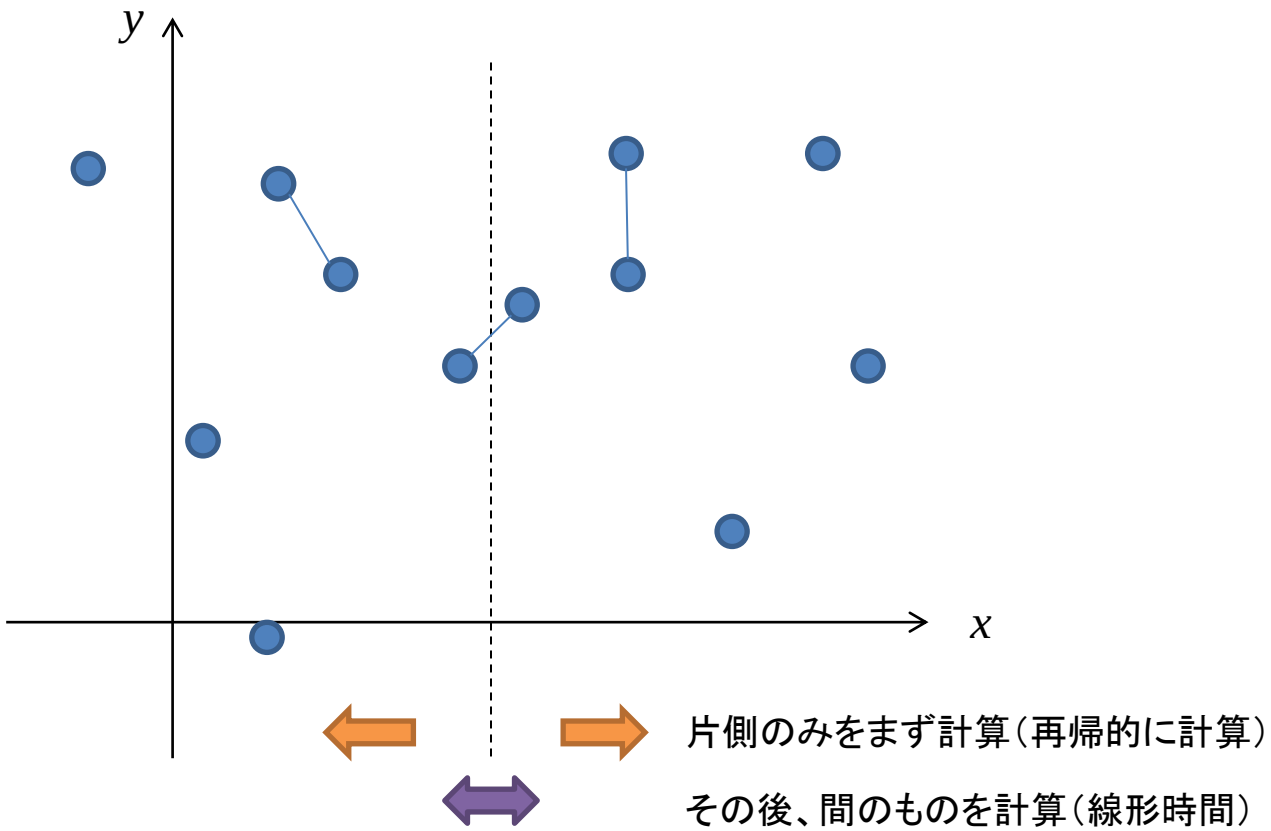
Tandem Repeat (4)　LCPを用いる

- LCPを用いると、
  - i 番目から繰り返し長 k のタンデムリピートがどれくらいの長さ続いているかを $O(1)$ で計算することが可能！
- 2つのテクニックの組み合わせると、最悪でも $O(n \log n)$
  - 分割統治に加えてLCP
- 実は、
  - 線形時間で計算する方法もある（Kolpakov & Kucherov '99）
- ただし、実際の実応用（生物学）では、
  - 完全一致以外の類似繰り返しを考慮する必要がある

	アルゴリズム	最悪計算量	平均計算量
1	超ナイーブ法	$O(n^3)$	$O(n^2)$
2	分割統治法	$O(n^2)$	$O(n \log n)$
3	LCP＋超ナイーブ法	$O(n^2)$	$O(n^2)$
4	LCP+分割統治法	$O(n \log n)$	$O(n \log n)$
5	LCP + factorization	$O(n)$	$O(n)$

類似アルゴリズム：2次元平面上の最近点ペアの探索 （1/2）

- 2次元平面上の最近点ペアの探索
  - ナイーブに全2点間を計算すると$O(n^2)$
  - 分割統治で$O(n \log n)$
    - 事前に x 軸に沿ってソートしておく

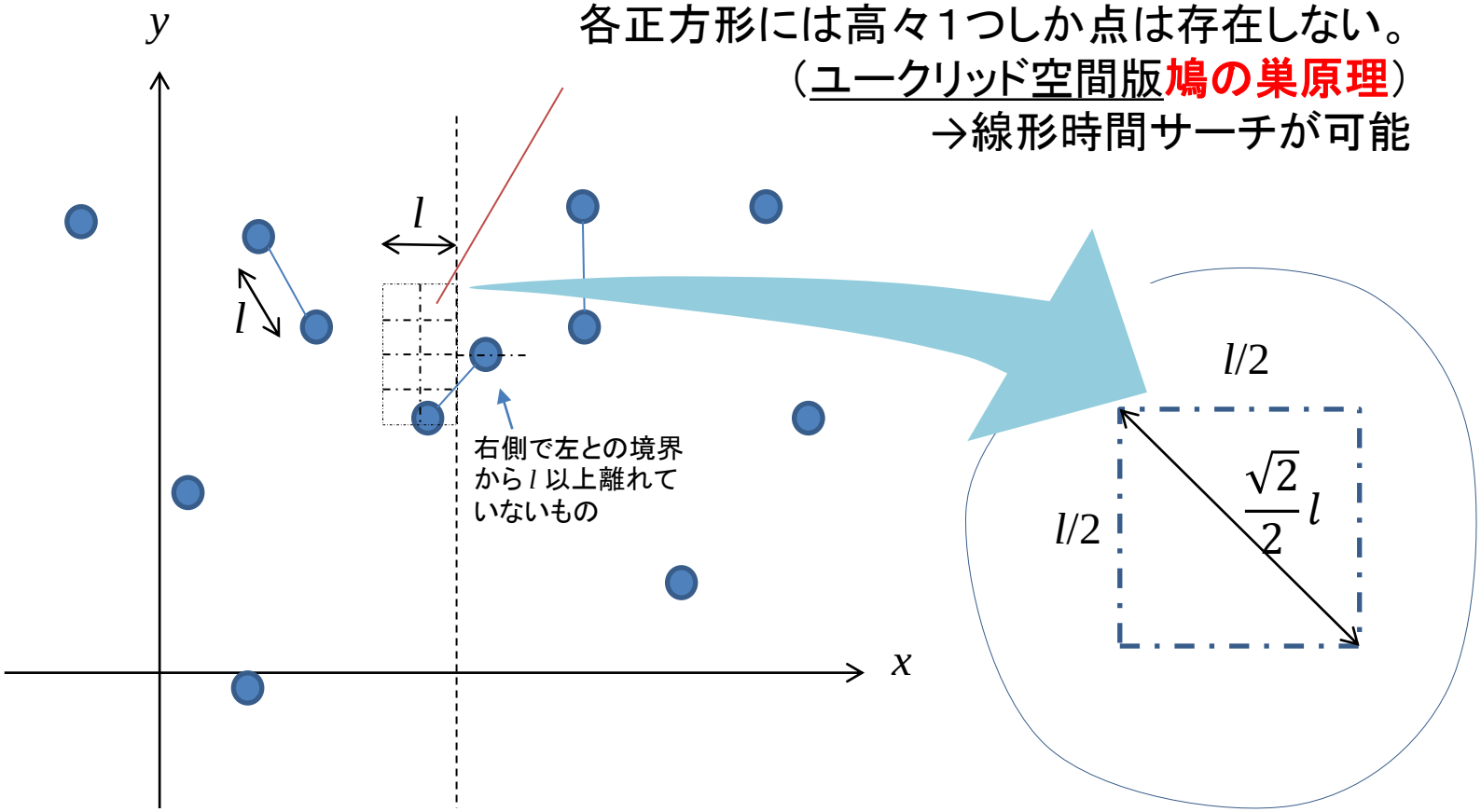


2次元平面上の最近点ペアの探索 （2/2）

両側の間の最近点ペア計算

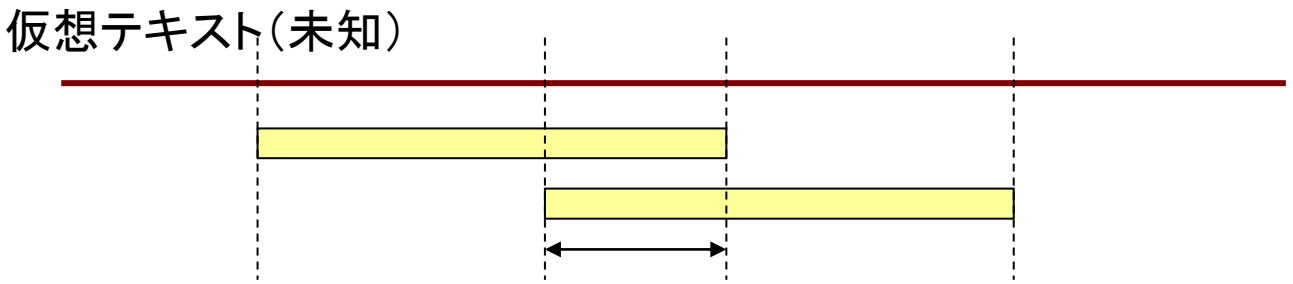
- 左側の最近点ペア間距離よりも小さいもののみ計算する
  - 事前にy軸に沿ってソートしておく（x軸のソートとは別に）

長さl以下のものはこの8つの正方形に入る。
各正方形には高々1つしか点は存在しない。
（ユークリッド空間版**鳩の巣原理**）
→線形時間サーチが可能



応用 8:　(All-pairs) Suffix-Prefix Matching (1)

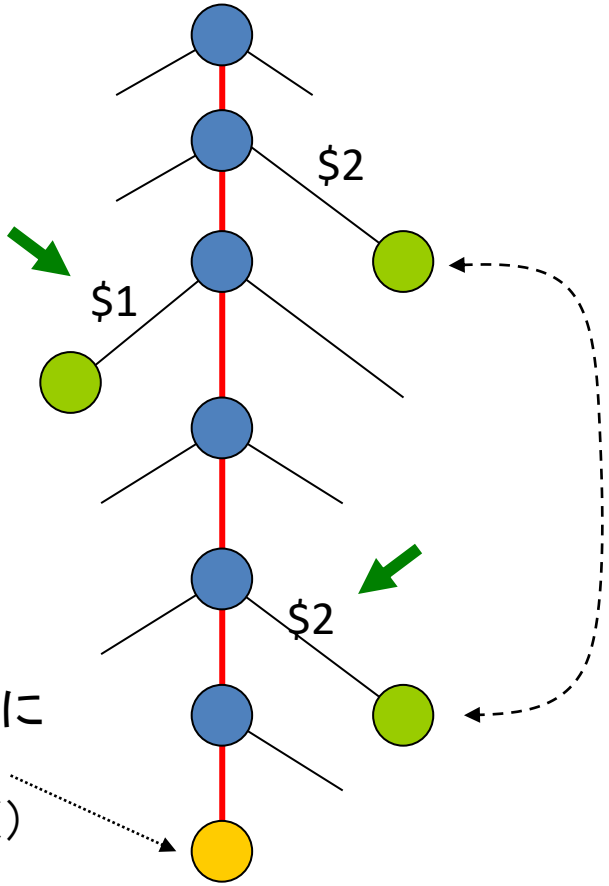
- ある2つ（以上）の文字列がどれくらい前後でオーバーラップしているかを知りたい
  - DNAアセンブリなどで必要になる



どれほど長く一致させることができるか？

(All-pairs) Suffix-Prefix Matching (2)

- 一般化接尾辞木を作成する
  - $O(n + \#\text{pairs\_to\_solve})$ で計算可能
  - 全対全の比較が一度に可能
- 現実にはエラーを考慮する必要がある



同じ文字列の$記号があれば
深い方を選ぶ
　＝よりオーバーラップが大きい

ある文字列全体に
相当する葉
（接尾辞ではなく）

は$記号のみからな
る枝で分かれている葉

接尾辞配列・接尾辞木の拡張・一般化

- 簡単な拡張(簡略化)
  - STのDAG化
  - SAの間引き
- 接尾辞木は非常に強力なので、文字列以外のデータに対しても同様の探索法を作りたいことがある
  - Circular Stringに対するST, SA
  - 文字の入れ替えを許すもの
    - *p*–suffix tree
    - *s*–suffix tree
  - 木構造に対するsuffix tree
  - 木構造に対するsuffix tree その2
    - BSuffix tree
  - 行列に対するsuffix tree
    - LSuffix tree
  - 空間上の点列に対するsuffix tree
    - Geometric suffix tree

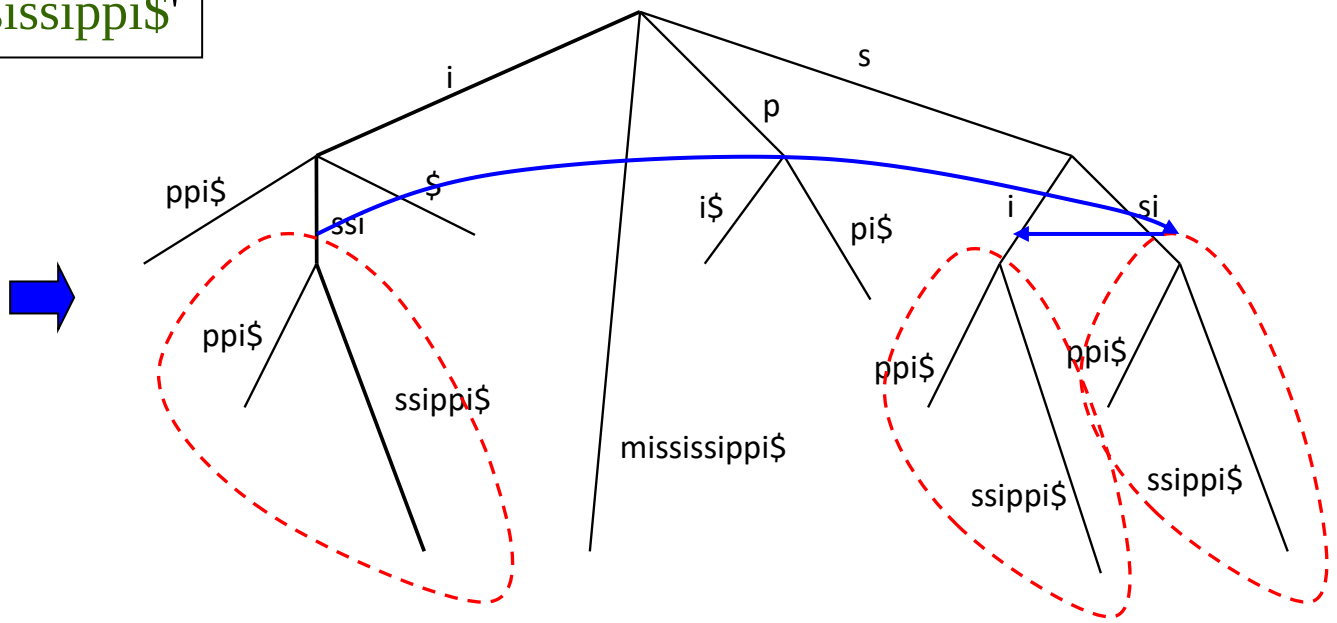
拡張 1： Suffix Link の応用：Suffix TreeのDAG化

- 接尾辞木の中を見ると、類似構造がある
  - それをまとめてsuffix treeを小さくできる
  - Suffix Linkをたどって、子供の数が同じであればまとめる
    - Suffix Link Treeを辿るだけでできるので、$O(n)$

Suffix tree of 'mississippi$'

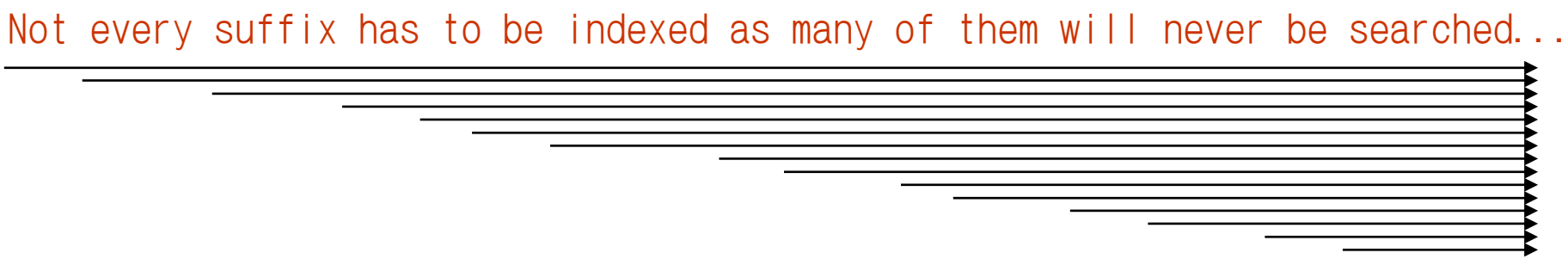
All the suffixes

mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$



拡張 2： ：Suffix Tree/Arrayの間引き

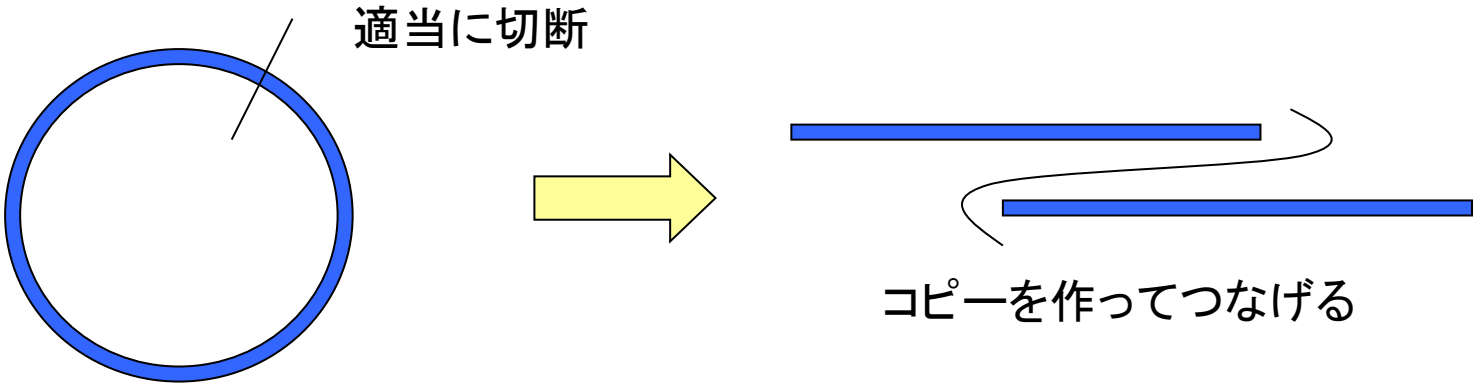
- 通常のテキストで区切りがしっかりしている場合はすべての接尾辞を扱う必要はない
  - 全部を作成して、間引きする
  - 一部の接尾辞集合からBentley-Sedgewick等の文字列用のアルゴリズムを用いる（接尾辞配列）
  - 単語を一つの文字だと思ってハッシュあるいは単純なテーブル（あるいはキーワード木）で数値に変換し、変換した数列に対し接尾辞木・接尾辞配列を作成する



注）他にも（もっと高度な）接尾辞木・接尾辞配列を圧縮する手法が多く開発されている

拡張 3： Circular String に対する接尾辞木・配列

- バクテリアのDNAなどは環状
  - 環状の文字列に対する接尾辞木・接尾辞配列が必要
- （仮想的に）二回繰り返した文字列を考えるだけ



Parameterized Matching [Baker '92]

- 2種類のアルファベット
  - 通常のアルファベット
  - パラメータ
    - permutation

p-match

$$Axyz\mathrm{B}yz = Ayzx\mathrm{B}zx \quad (x \to y, y \to z, z \to x)$$

$$\Pi = \{x, y, z\} \quad \Sigma = \{A, B\}$$

- Encoding for *p*-Strings (*p*-Encoding)
  - 前出の同一パラメータまでの距離に変換
    - 0 for left-most parameters
  - このコーディングが同じであれば*p*-match

$$\mathrm{prev}(A\overbrace{xyz}\mathrm{B}\underbrace{yz}) = \mathrm{A000B33}$$

- 効能
  - 学生のレポートが他人のレポートのコードの変数を改変しただけのコードだった時にそのチェックに使うことができる。
  - ソフトウェアを発注した時に、行数でお金を払うが、単なるコピーはもちろん、変数だけを改変したコードにはお金を払いたくない、というときのチェックができる。
  - RNA構造解析

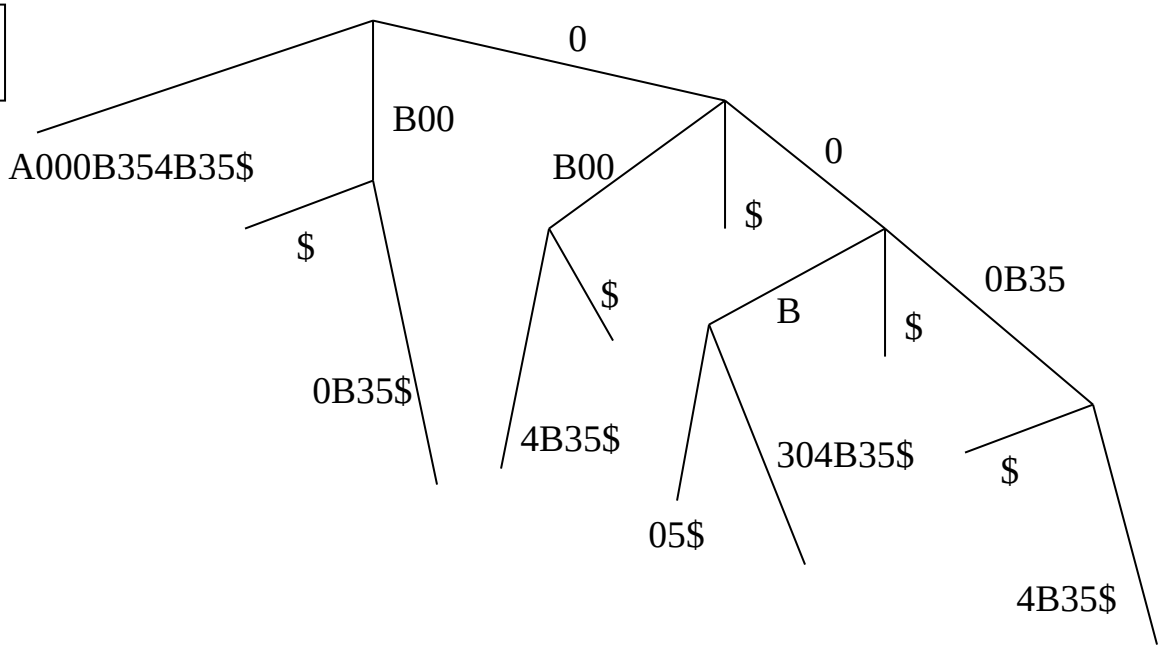
拡張4：Parameterized Suffix Tree

- *p*–stringに対する接尾辞木
  - 通常の接尾辞木とは異なり、*p*–suffix のsuffixは*p*–suffixとは限らない
  - 構成アルゴリズム
    - $O(n(\log|\Sigma|+\log|\Pi|))$　[Kosaraju ’95, Shibuya ’04]

p-Suffix tree of ‘A*xyz*B*yxz*B*xy*$’

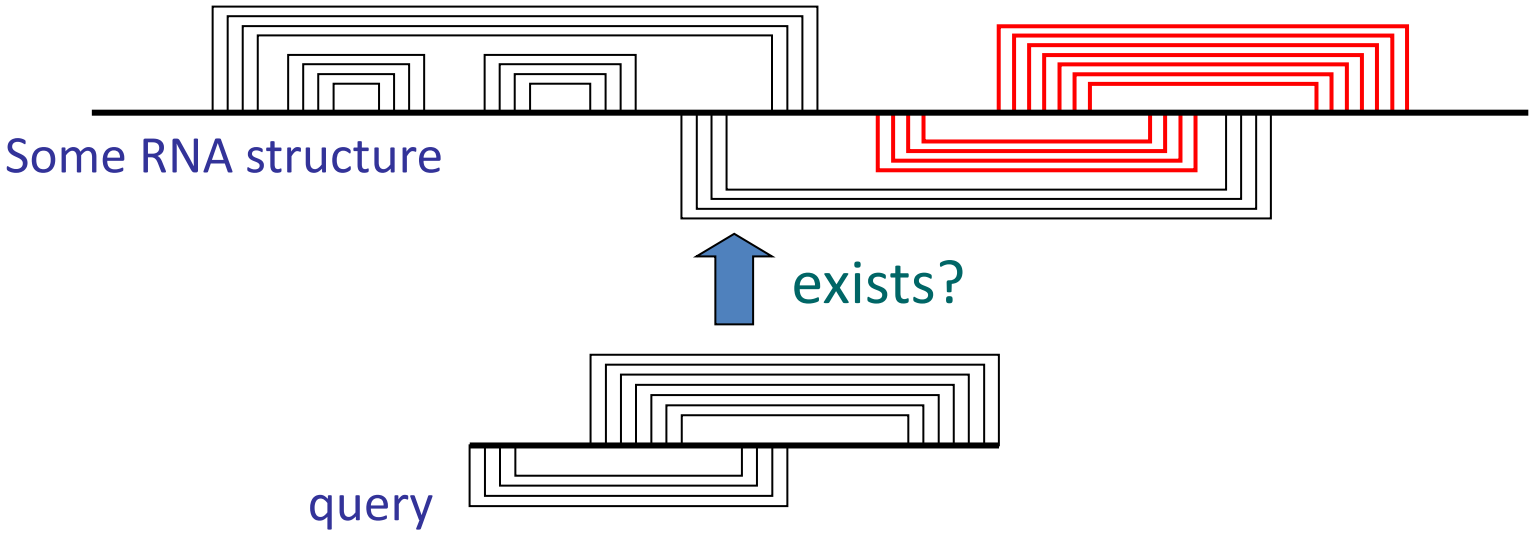
Encoded suffixes

A000B354B35$
000B354B35$
00B3**0**4B35$
0B**0**04B35$
B00**0**B35$
000B35$
00B**0**5$
0B0**0**$
B00$
00$
0$



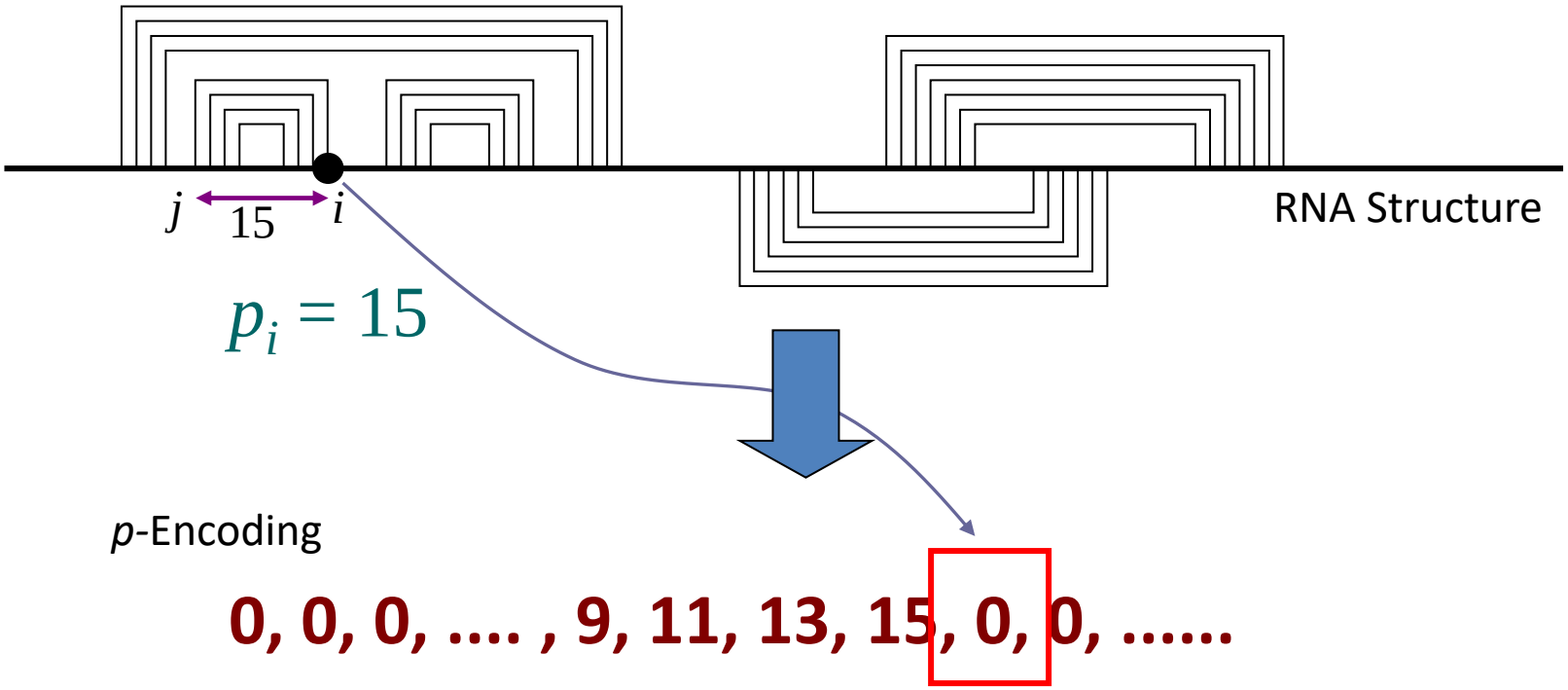
RNA 構造比較

- RNA構造の部分構造で全く同じ部分を探したい



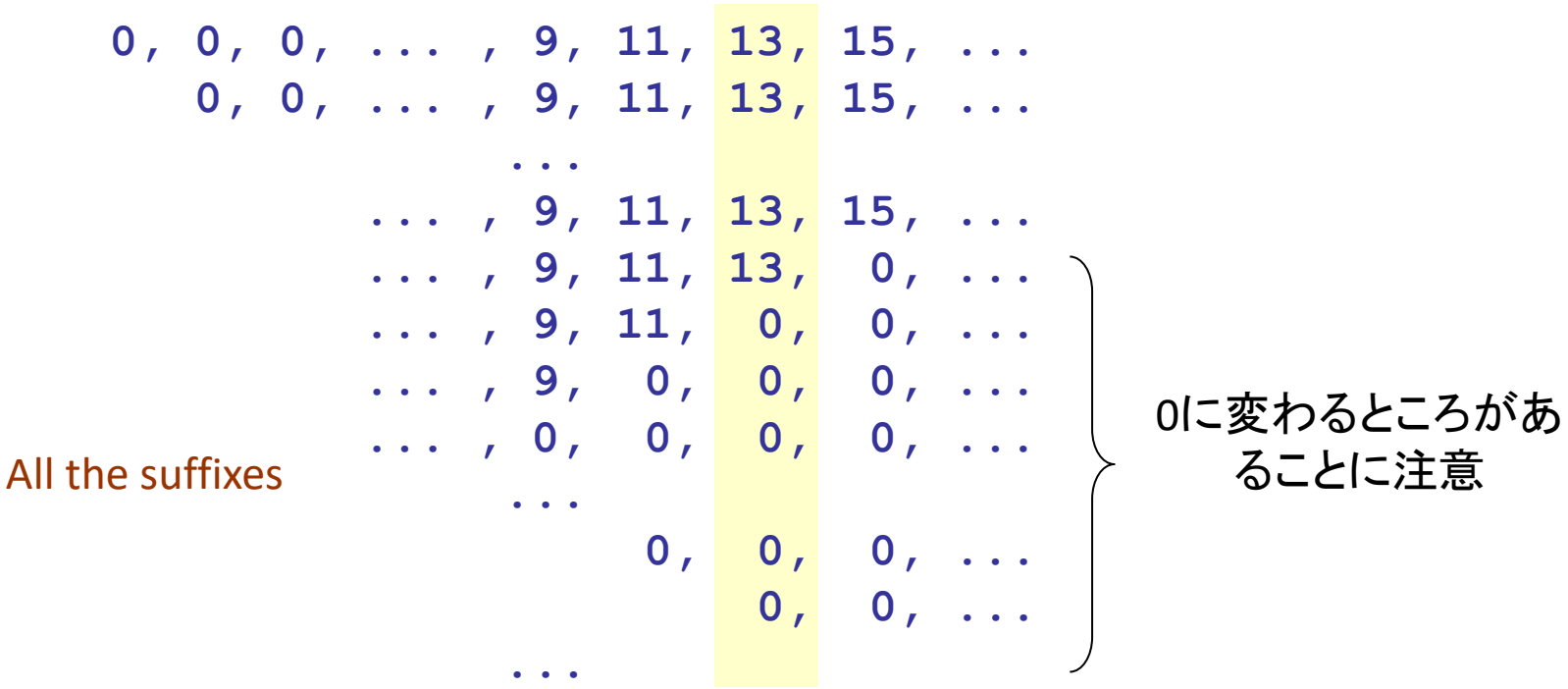
RNA構造をp-stringで表現

- i 番目と j 番目が結合していたら $p_i = i - j$ （ただし $j<i$).
- それ以外の場合 $p_i = 0$.



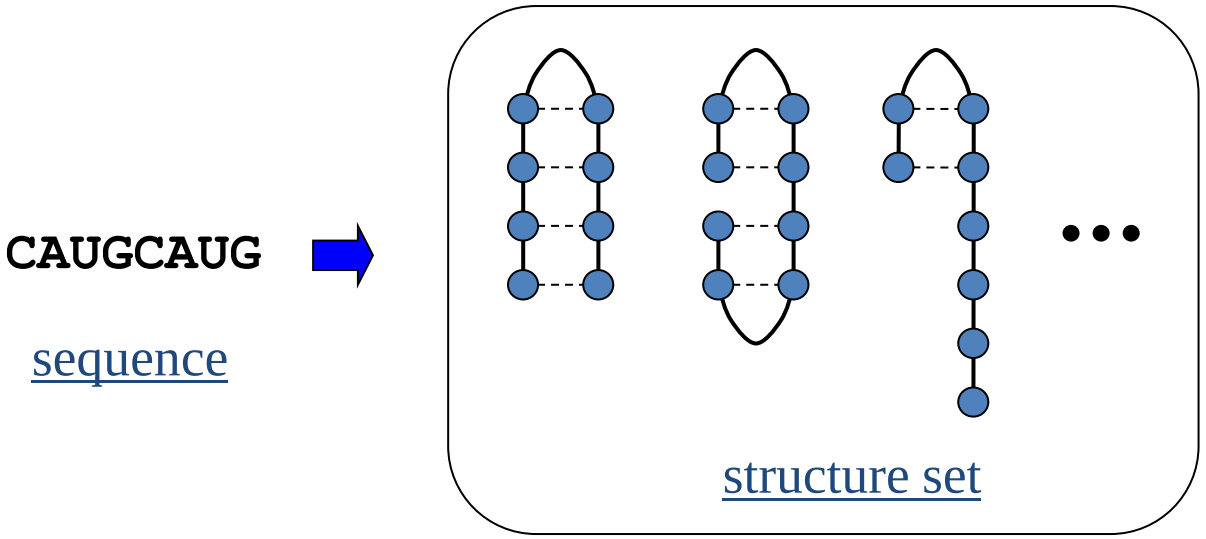
Then...

- あとはp-suffix treeを作成するだけ



拡張5：RNAのマッチング　その2

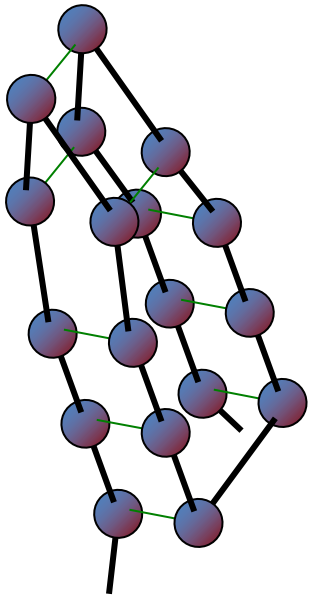
- あるRNA配列がとりえる構造の集合が同じであるような部分文字列を探したい

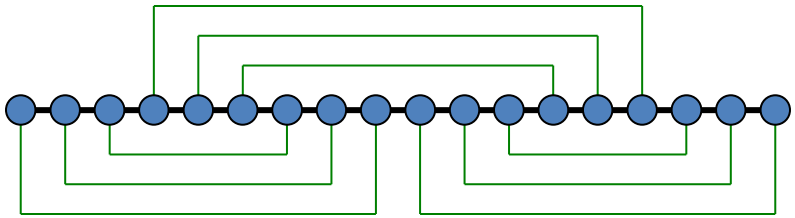


例

AUGUACGUAAUCGGCGUGUCCCGUUAUCCGUGAG
UCGGACUUAUAAUAUCGUAUGGCCGAGCCUGCCU
CAUGUGCCACGUACGCCGUAGUGACACAUCCGCG
UCCGCGUAGCGAAUUACAUUGUCUAAUCGAUAGC
GACUAACCGCUGACUGUAAGCGCUCCGCGGUCAA

substring 1: **AUAUCGUAUGGCCGAGCC**
substring 2: **CGCGUAGCGAAUUACAUU**





One of the structures

Structural String (*s*-String)

[Shibuya '04]

- *p*-stringのパーミュテーションに制限をつける
- 直前の相補文字への距離も用いる

RNA sequence : **AUAUCGUAUGGCCGAGCC**

prev(*i*) : 00220035241713754 1

cmpl(*i*) : 01110141156121631 2

[s-match]

$$\mathrm{A}xy\mathrm{B}zwyzwx = \mathrm{A}zw\mathrm{B}yxwyxz \quad (x \to z, z \to y)$$

$$\Sigma = \{\mathrm{A}, \mathrm{B}\}, \ \Pi = \{x, y, z, w\}$$

Complementary pairs: $\{x,y\}$, $\{z,w\}$

$$x \to y \Leftrightarrow \mathrm{complement}(x) \to \mathrm{complement}(y)$$

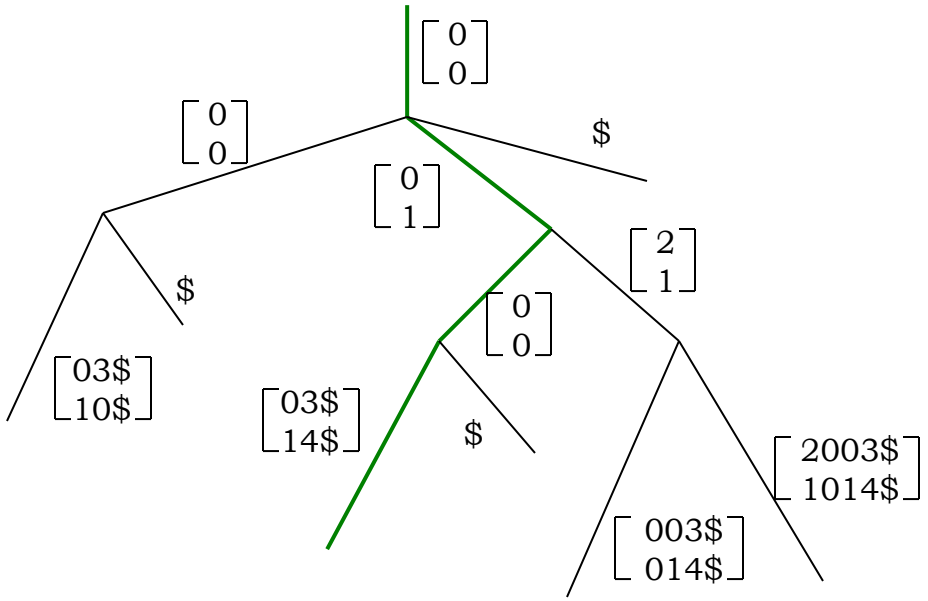
s-Suffix Tree

- *s*-suffix tree
  - 同じように*s*-suffixに対する接尾辞木
  - $O(n\ (\log p + \log q))$で作成可能 （p, q: アルファベットサイズ）

s-Suffix tree of ‘**AUAUCGU$**’

s-Encoded suffixes

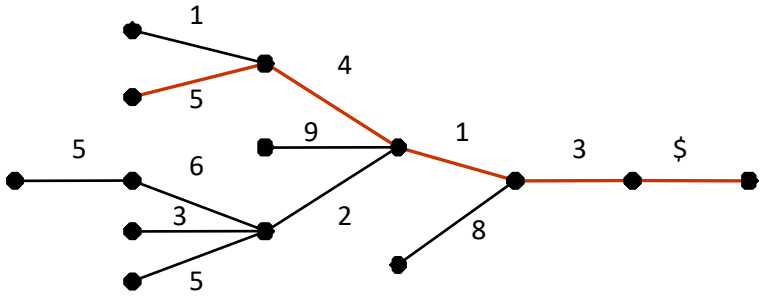
0022003$
0111014$
002003$
011014$
00003$
01014$
0003$
0010$
000$
010$
00$
00$
0$
0$



拡張6: 木に対する接尾辞木

- Target Tree
  - 枝は文字を持つ
  - 葉からルートへの文字列を考える
  - 表せる文字列のサイズは$O(n^2)$　　(n: size of the tree)
- 木に対する接尾辞木
  - 表されている文字列に対する一般化接尾辞木に相当するもの
  - $O(n)$ algorithm [Breslauer '98, Shibuya '03]
  - より小さく保持したデータ構造としてxbw [Ferragina, Manzini, 2009]がある
    - FM-indexの技術を用いて簡潔構造として表現
- 効能
  - 木の部分マッチングのアルゴリズムの(理論的な)サブルーチンとして使う
    - その後、よりよいアルゴリズムが出てきて今は使われていない
  - オートマトンの最小化
  - Tree kernel [Kimura et al. 2011]
  - 木構造の中から頻出パスを探せる

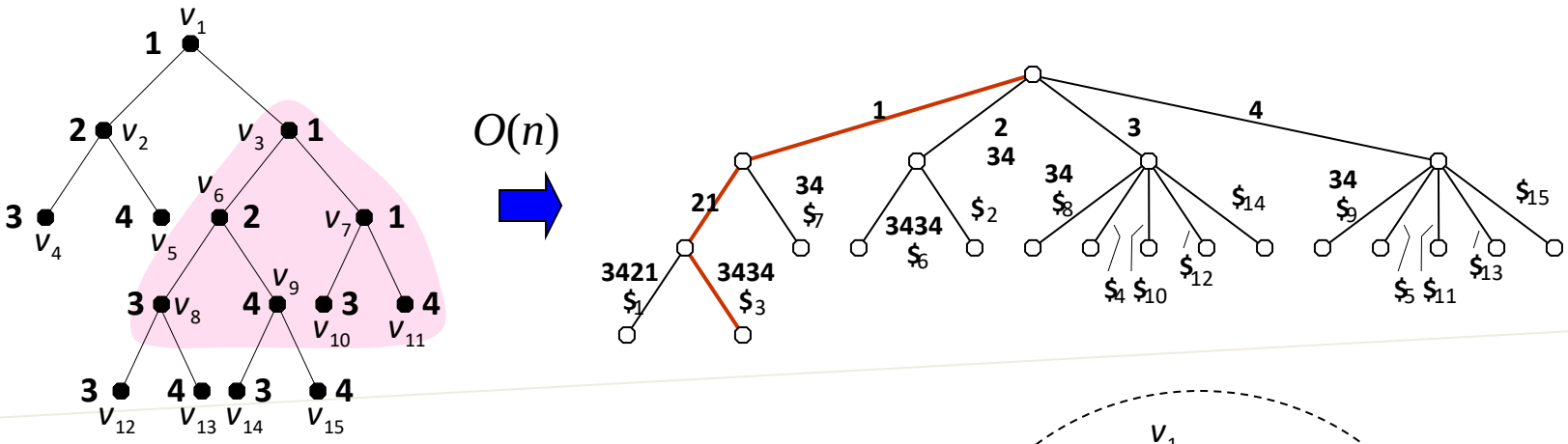
A tree representing
7 strings
Size: **13**
Sum of the represented
string sizes : **33**



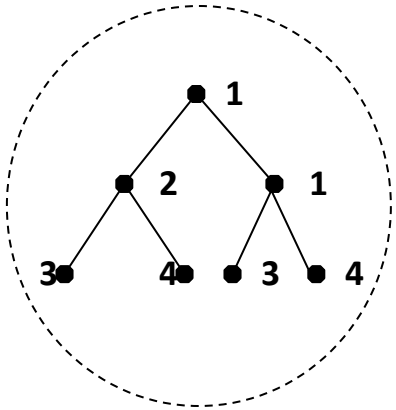
1313$
5413$
913$
56213$
3213$
5213$
83$

拡張7: 木に対する接尾辞木 （2）： BSuffix Tree

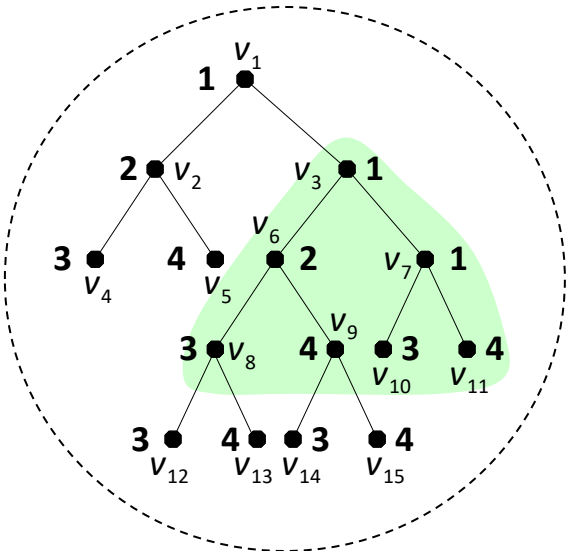
- 順序付二分木の中の完全二分木である部分木を表したtrie
  - 木の中から完全二分木である部分木を探す



Does



exist in

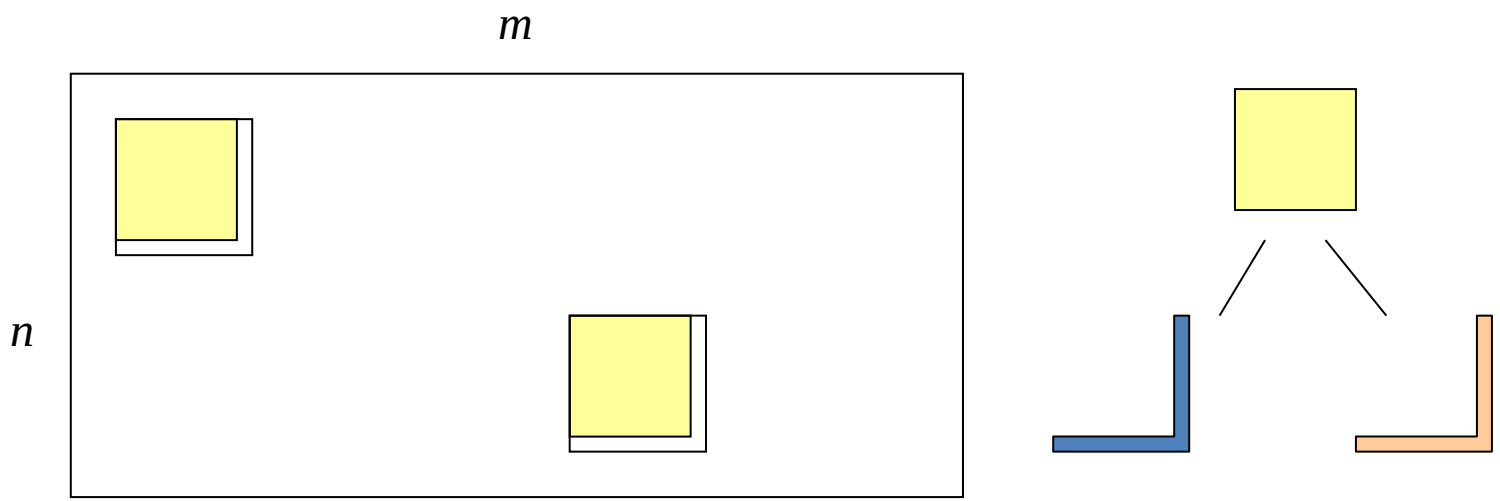


?

拡張8： LSuffix Trees (ISuffix Trees)

[Giancarlo '93]

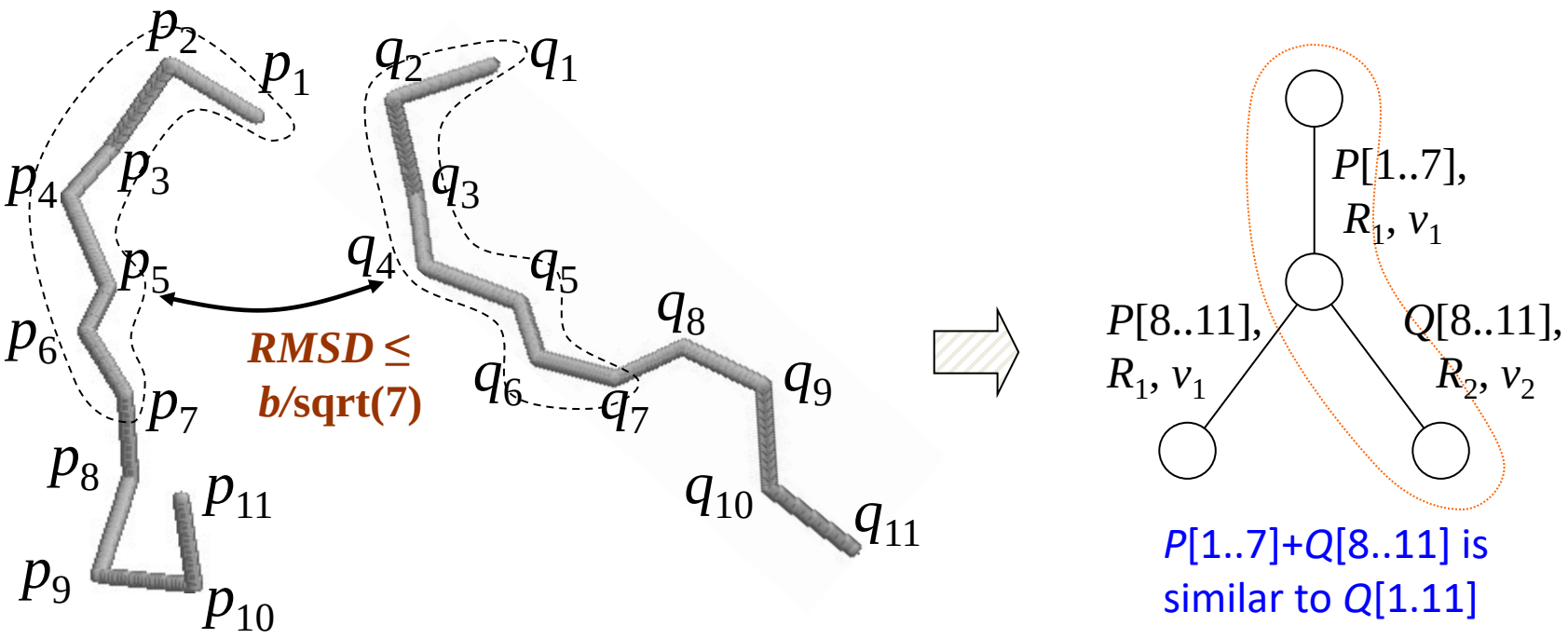
- $n \times m$行列上の部分正方行列をすべてtrieで表したもの
  - Farachのアルゴリズムの応用で$O(nm)$ [Kim & Park '99]
- 画像処理への応用も
  - トリミング画像検索
- k次元への拡張も
  - さらにp–suffix treeと組み合わせた一般化もあり



拡張9：Geometric Suffix Tree

[Shibuya J.ACM 2010]

- Suffix treeを3次元構造検索へ拡張
  - RMSDがincrementalに計算できることを利用

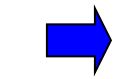


Block Sorting 圧縮 [Burrows, Wheeler '94]

- BWT (Burrows-Wheeler Transform)
  - Suffix Array と極めて関連が深い （FM-Index）
- MTF（move to front）coding
  - ここまでは全く圧縮されていない状態
- これらを行った後、通常のハフマン符号・算術符号等を用いてさらに圧縮
- 処理するのに巨大すぎる場合はブロック化→ブロックソーティング
  - BW変換するためのメモリ・計算時間に限界

cf.

```
 0: mississippi$
 1: ississippi$
 2: ssissippi$
 3: sissippi$
 4: issippi$
 5: ssippi$
 6: sippi$
 7: ippi$
 8: ppi$
 9: pi$
10: i$
```



ソート

```
10: i$
 7: ippi$
 4: issippi$
 1: ississippi$
 0: mississippi$
 9: pi$
 8: ppi$
 6: sippi$
 3: sissippi$
 5: ssippi$
 2: ssissippi$
```

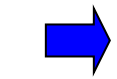
接尾辞配列

BWT (Burrows-Wheeler Transform)

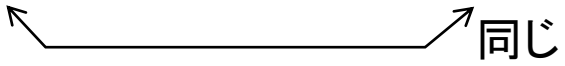
- 接尾辞配列 : $SA[0..n]$
  - ただし、文字列はサイクル化したものを考える
    - 文字列の最後に$を入れておけば、通常のSAと特に違いはない
- $BWT[i] = T[\mathrm{SA}[i]-1]$

サイクル化した全接尾辞

```
 0: mississippi$
 1: ississippi$m
 2: ssissippi$mi
 3: sissippi$mis
 4: issippi$miss
 5: ssippi$missi
 6: sippi$missis
 7: ippi$mississ
 8: ppi$mississi
 9: pi$mississip
10: i$mississipp
11: $mississippi
```



BWT	接尾辞配列
10: i	11: $mississippi
9: p	10: i$mississipp
6: s	7: ippi$mississ
3: s	4: issippi$miss
0: m	1: ississippi$m
11: $	0: mississippi$
8: p	9: pi$mississip
7: i	8: ppi$mississi
5: s	6: sippi$missis
2: s	3: sissippi$mis
4: i	5: ssippi$missi
1: i	2: ssissippi$mi



BWTの性質

- 似た文字がかたまって出現する
  - 似た文字列の前は同じ文字が出現しやすいから！
  - この性質を利用して圧縮するとかなり小さくできる
    - MTF変換＋算術符号等
- この文字列からもとの文字列を復元できる！

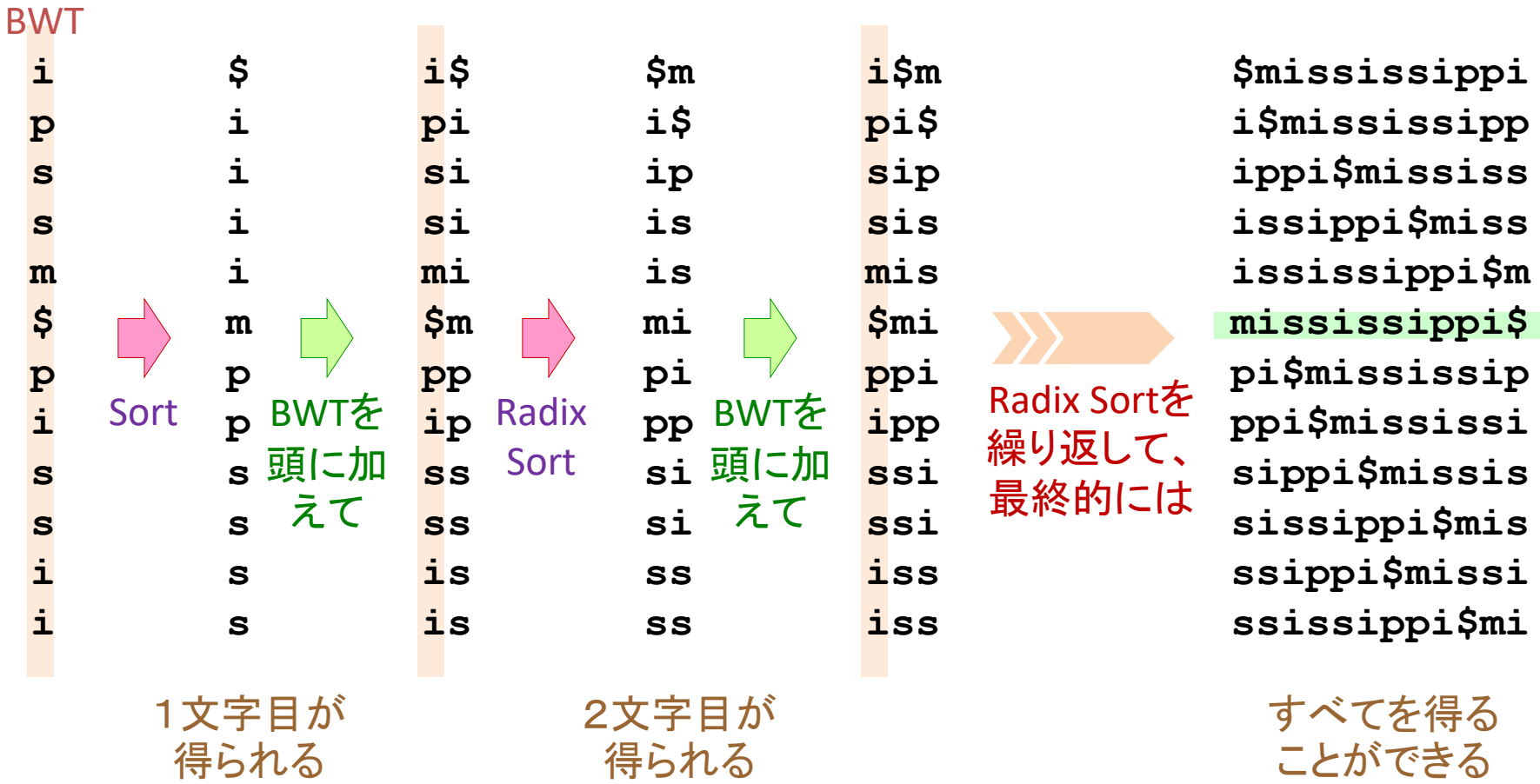
mississippi$

復元

BWT		接尾辞配列
10:	i	11: $mississippi
9:	p	10: i$mississipp
6:	s	7: ippi$mississ
3:	s	4: issippi$miss
0:	m	1: ississippi$m
11:	$	0: mississippi$
8:	p	9: pi$mississip
7:	i	8: ppi$mississi
5:	s	6: sippi$missis
2:	s	3: sissippi$mis
4:	i	5: ssippi$missi
1:	i	2: ssissippi$mi

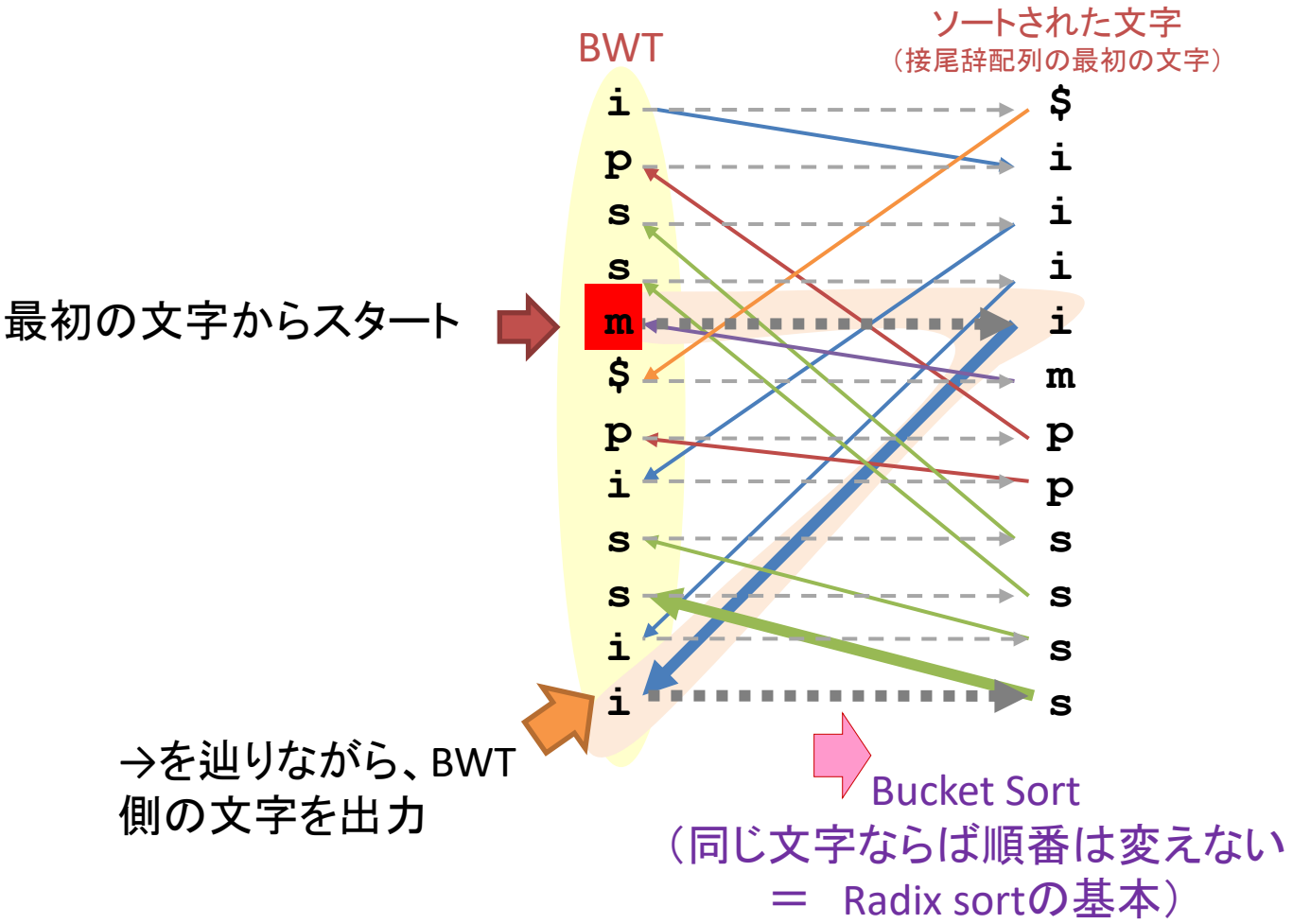
BWTの復元

- n 桁のRadix Sortを行うだけ
  - ただし、そのままともにやると計算量が $O(n^2)$
  - 実は$O(n)$にできる！



BWTの線形時間復元

- （一桁分の）ソートの動きを逆に辿っていく
  - 最初の文字がどれに相当するか覚えておく
    - ただし、覚えてなくても $ を探してそこから始めればOK



MTF(move to front) Coding

- 同じ文字が前回出現してから、何種類の他の文字が出現したか？
  - 初出の文字は適当に処理
  - BWT後の文字列を変換すれば偏りのある数字列になる
    - 小さい数字が多く現れる
    - 算術符号その他の適当なアルゴリズムで圧縮すれば、かなり小さくできる
  - デコードは簡単
  - 計算量
    - バランス木で表現すれば$O(n \log s)$

テキスト　**adbdbaabacdadc**

変換した数字　**03211201133212**

変換表

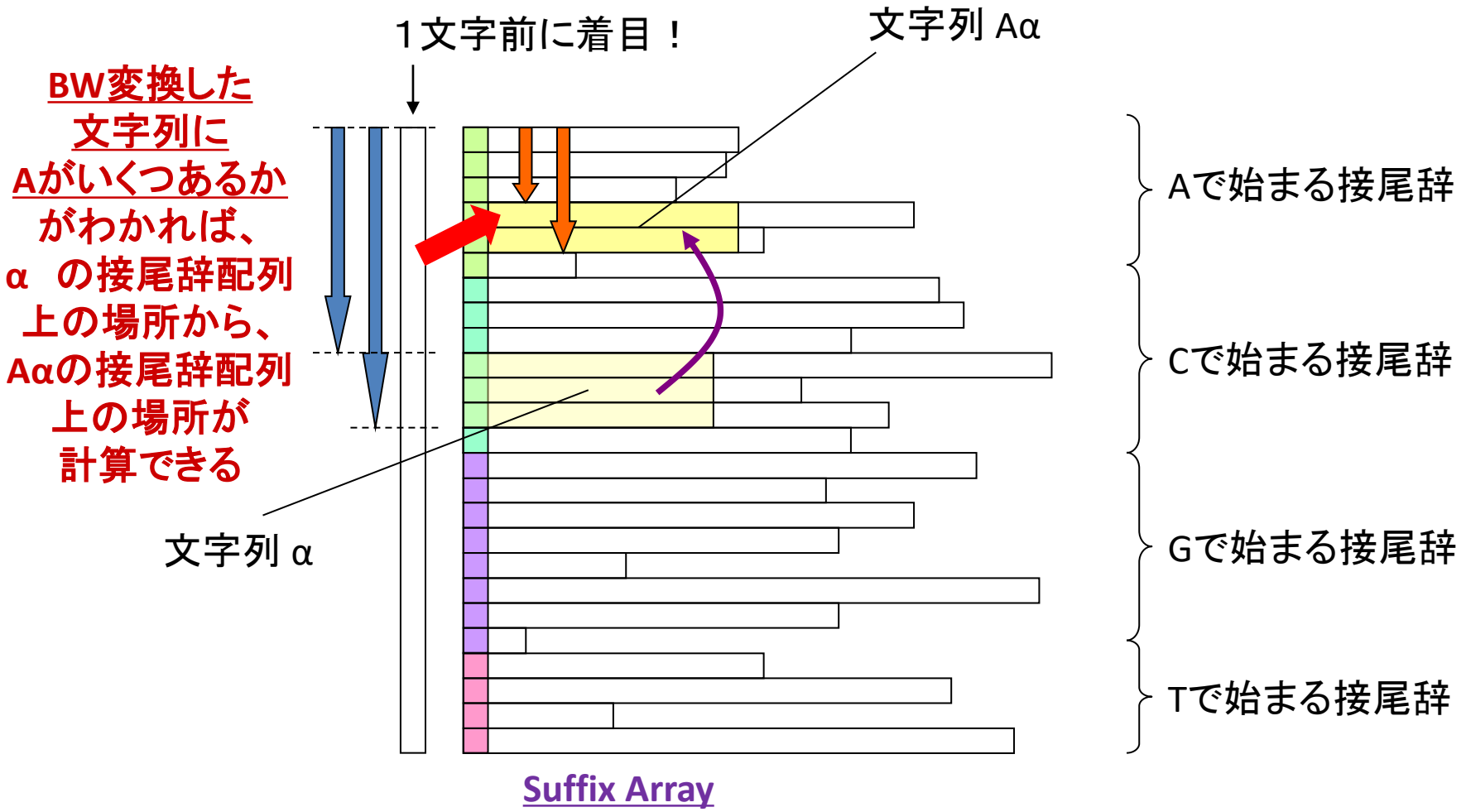
0	**a**adbdb**a**abacdadc
1	bba**db**db**ba**bac**d**ad
2	cc**b**aa**a**ddddb**a**c**c**a
3	d**d**ccccccccc**cd**bbbb

出来の悪いウィンドウズのメニューみたいな

適当

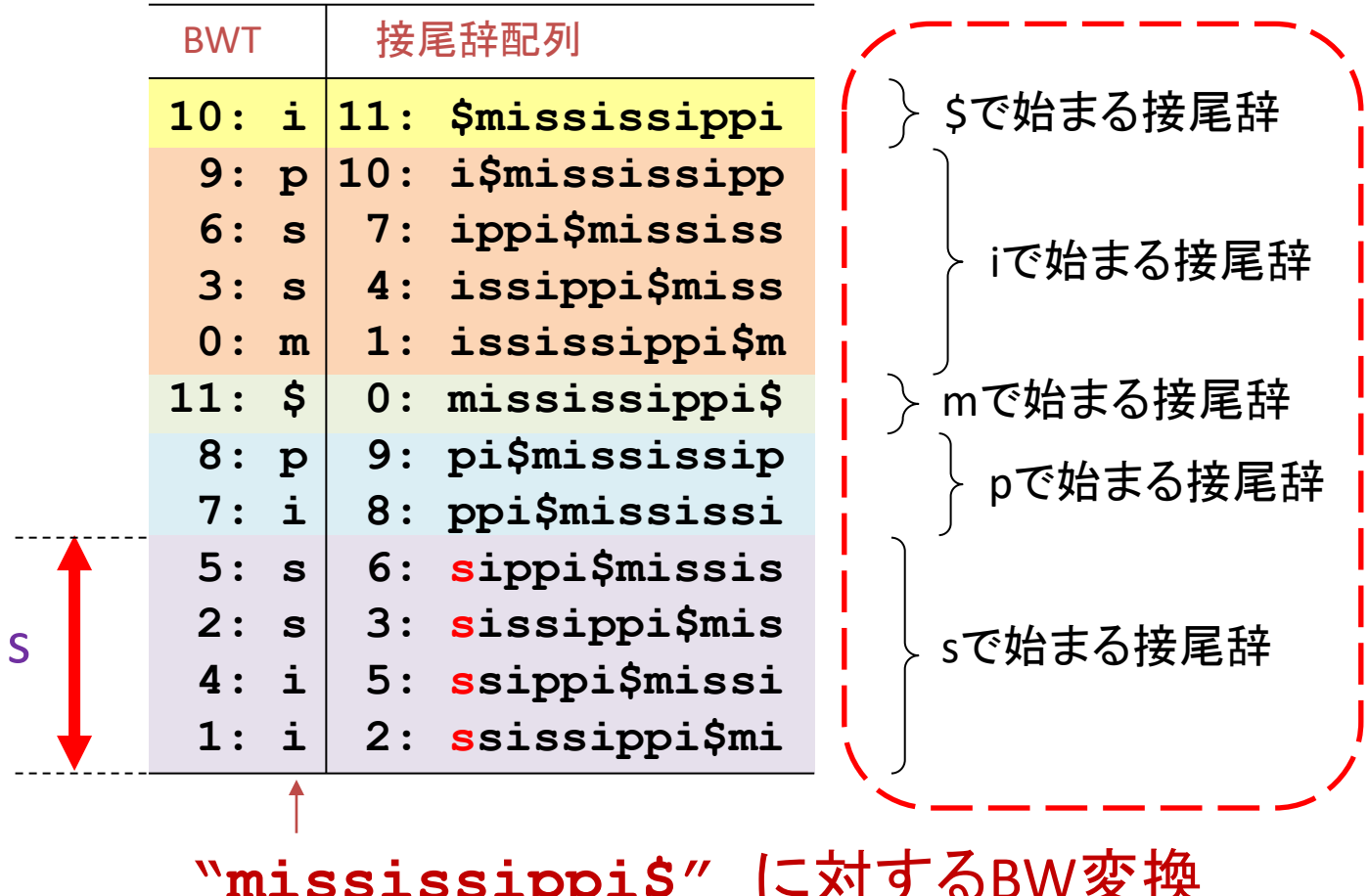
BW変換による索引： FM-Index

- BW変換を用いて検索もできる！ [Ferragina Manzini, 2000]
  - 逆方向検索： 接尾辞木のWeiner Algorithmの逆suffix linkを辿ることに相当する



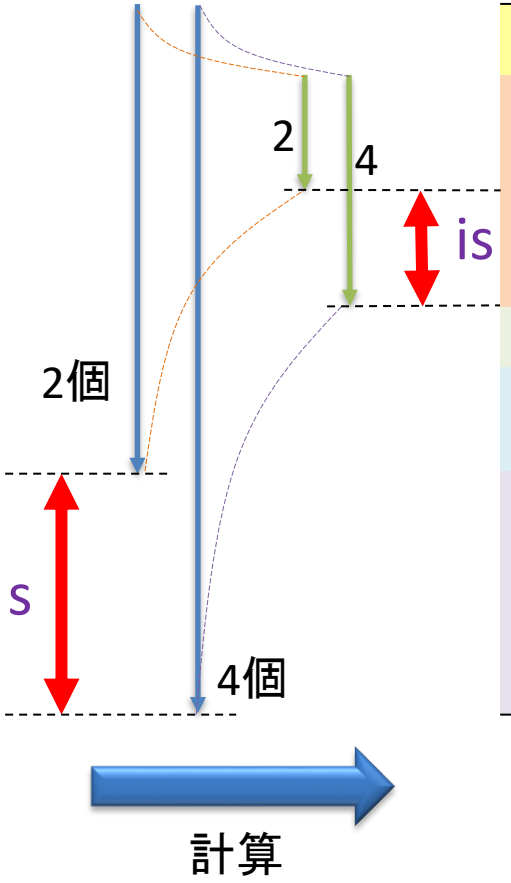
例 (1/4)

- 後ろから検索する！
- “ssis”の最後の1文字“s”は？
  - これは覚えておいたものをそのまま使う



例 (2/4)

- “ssis” の後ろ2文字 “is” は？
  - BW変換中で “i” の数を数える
    - 実際にはwavelet treeを用いる

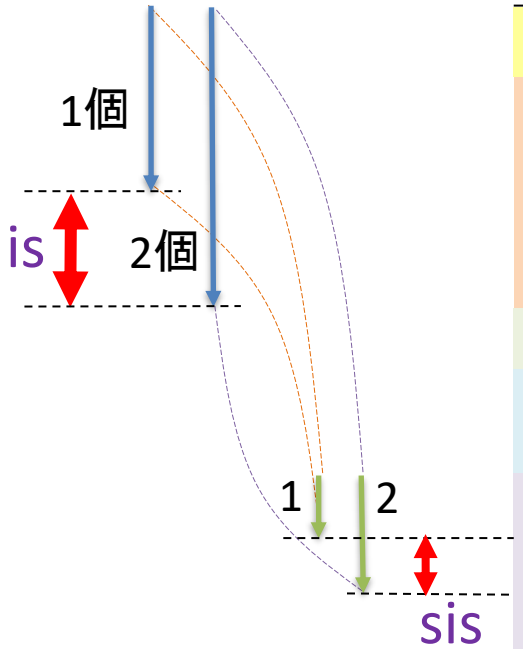


BWT	接尾辞配列	
10: i	11: $mississippi	$で始まる接尾辞
9: p	10: i$mississipp	iで始まる接尾辞
6: s	7: ippi$mississ	
3: s	4: issippi$miss	
0: m	1: ississippi$m	
11: $	0: mississippi$	mで始まる接尾辞
8: p	9: pi$mississip	pで始まる接尾辞
7: i	8: ppi$mississi	
5: s	6: sippi$missis	sで始まる接尾辞
2: s	3: sissippi$mis	
4: i	5: ssippi$missi	
1: i	2: ssissippi$mi	

“mississippi$” に対するBW変換

例 (3/4)

- “ssis” の後ろ3文字 “sis” は？
  - BW変換中で “s” の数を数える



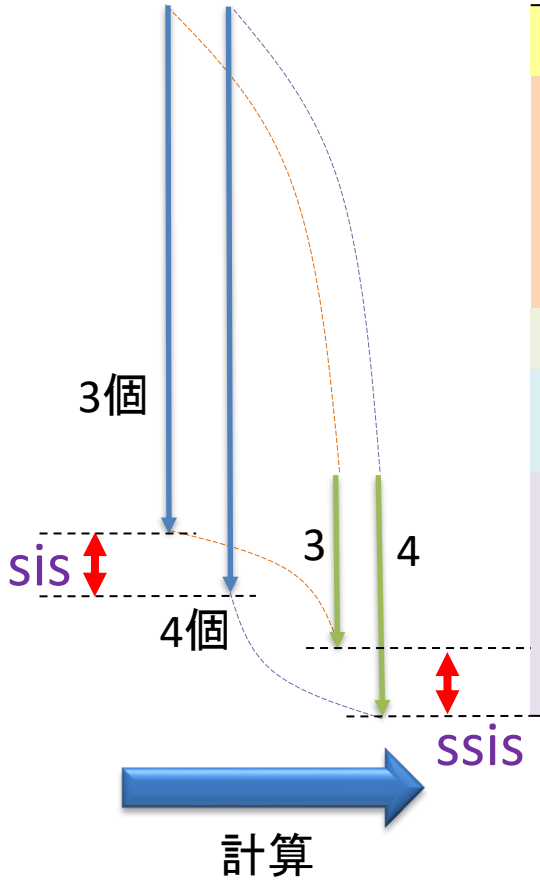
BWT	接尾辞配列	
10: i	11: $mississippi	$で始まる接尾辞
9: p	10: i$mississipp	iで始まる接尾辞
6: s	7: ippi$mississ	
3: s	4: issippi$miss	
0: m	1: ississippi$m	
11: $	0: mississippi$	mで始まる接尾辞
8: p	9: pi$mississip	pで始まる接尾辞
7: i	8: ppi$mississi	
5: s	6: sippi$missis	sで始まる接尾辞
2: s	3: sissippi$mis	
4: i	5: ssippi$missi	
1: i	2: ssissippi$mi	

計算

“mississippi$” に対するBW変換

例 (4/4)

- “ssis”は？ （これが求めるもの）
  - BW変換中で“s”の数を数える

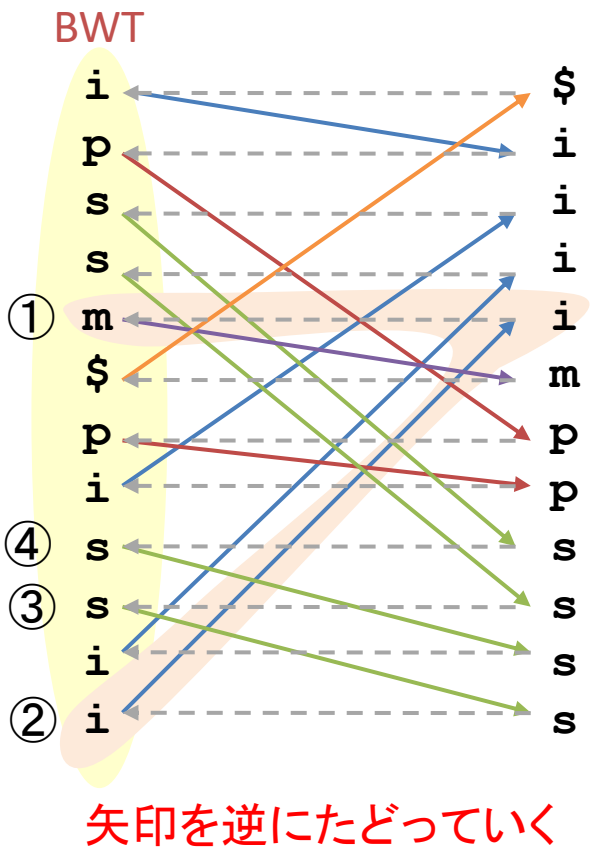


BWT	接尾辞配列	
10: i	11: $mississippi	$で始まる接尾辞
9: p	10: i$mississipp	iで始まる接尾辞
6: s	7: ippi$mississ	
3: s	4: issippi$miss	
0: m	1: ississippi$m	
11: $	0: mississippi$	mで始まる接尾辞
8: p	9: pi$mississip	pで始まる接尾辞
7: i	8: ppi$mississi	
5: s	6: sippi$missis	sで始まる接尾辞
2: s	3: sissippi$mis	
4: i	5: ssippi$missi	
1: i	2: ssissippi$mi	

“mississippi$” に対するBW変換

FM-Index (3)

- 位置を知るには？
  - BW−変換されたテキストの上のある位置が元のどの位置に対応するのか（＝すなわちsuffix arrayの値）は、そこからdecodeを行えばよい
    - BWTの復元の要領で
    - 接尾辞が復元される（＝その長さがわかる＝位置がわかる）
    - ただし、それでは $O(n)$ の時間がかかる
  - 間引きされたsuffix arrayを持っておくとよい
    - z ごとに間引くと、z 倍のオーバーヘッドがかかる



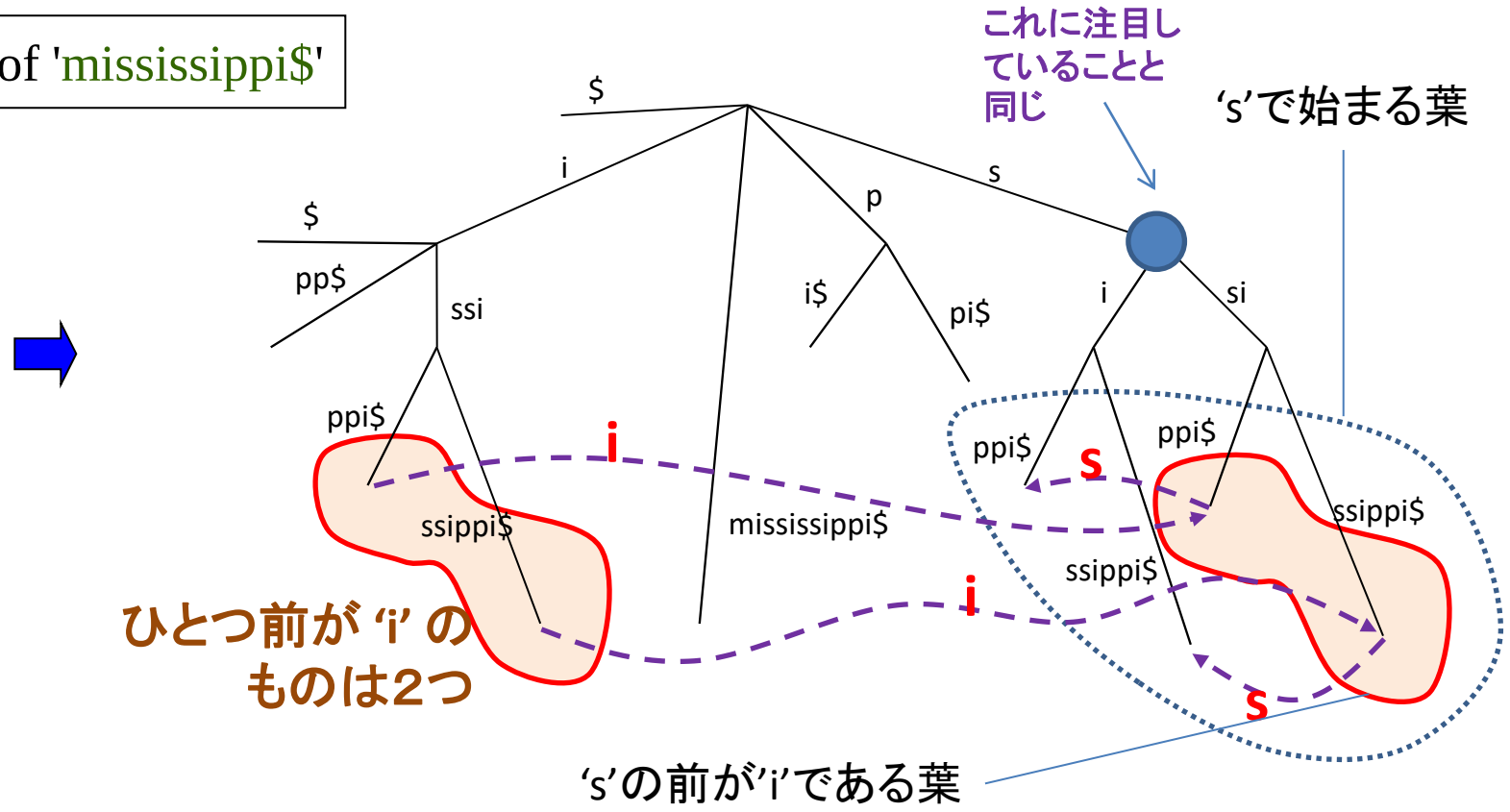
FM-Index (4) : WeinerのPrefix Linkと検索

- 葉のSuffix link（群）を逆にたどる（すなわちPrefix linkを辿る）と逆順の部分文字列になる、ということと対応している

Suffix tree of 'mississippi$'

mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$

All the suffixes



‘s’ の前が ‘i’ であるものを探す

FM-Index (5) : 検索に必要なデータ構造

- この時、BW変換した文字列以外に必要なもの：
  - なんと、接尾辞配列、元の文字列は保持しなくてよい！！！
    - すなわち、**以下の補助データ構造のサイズが十分小さければ、**これは極めてコンパクトな索引構造として利用することができる！
    - さらに、（遅さを気にしなければ）BW変換した文字列は圧縮可能！
      - ただし、任意の i に対し i 文字目に、（**圧縮率のために速さを犠牲にしたとしても**）何らかの方法でアクセス可能な圧縮方法でないといけない
  - 各文字が何文字ずつあるかを覚えておくことで、
    - 次のindexを計算できる
    - 逆に、indexからそれに対応する文字が何か計算できる
  - BW変換した文字列$BW[0..n]$上で（多少遅くてもよいので）
    - $BW[0..i]$に文字xがいくつあるか

  が計算可能な（なるべく小さな）データ構造

（**Rank/Select データ構造**）

Rank / Select データ構造

- 3つの操作
  - access(A, i) = $A[i]$
  - rank$_c$(A, i) = #j s.t., $A[j]$=c, j<i
  - select$_c$ (A, i) = j s.t., rank$_c$(A, j) = i
- ナイーブに表を持つと
  - access
    - 元の文字列を持っているだけでOK
  - rank
  - select
    - いずれも、(文字の種類数)×(文字列長)×(整数のサイズ)のサイズの表で、constant timeを(線形時間の前処理で)実現できる
    - 片方がconstant timeでできれば、他方は二分探索で$O(\log n)$が可能

	a	b	c
a	1	0	0
b	1	1	0
c	1	1	1
b	1	2	1
c	1	2	2
c	1	2	3

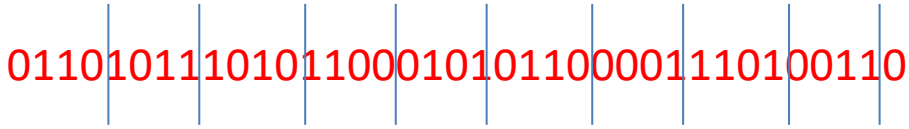
A

ナイーブなrank表

が、これではBW変換の補助データ構造のサイズが
通常の接尾辞配列のアルファベットサイズ倍、という
巨大なものになってしまう！(それではほとんど意味がない)

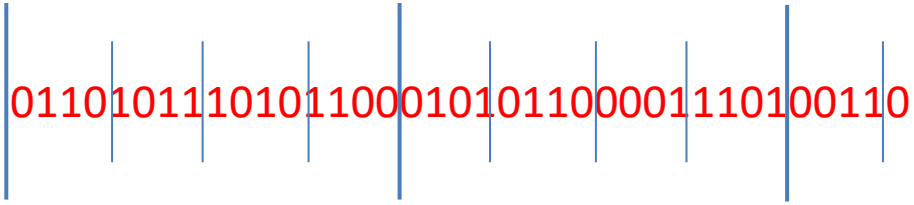
小さくする方法： ブロック化による間引き

- 01列の場合
  - wビットずつに分割して、wビットずつの間引きされたrank結果は覚えておく
  - 2^w種類のすべてに関してrankの結果を計算し、保管する
    - RMQのアルゴリズムと少し類似
  - すると必要なメモリは
    - $\binom{n}{w} \cdot \log n + 2^w \log w$ bit
    - $w = \log n$ とすると これは $n + n \log\log n$ bit
      - 接尾辞配列（$n \log n$ bit）よりは小さそう
      - が、BW変換後の文字列（n bit）よりは大きい
  - rankの計算時間は constant timeで可能



ブロック化による間引き　その2

- 先ほどのブロック化では、間引いたrankはすべての桁を保持していた（＝大きな数字を覚えていて無駄）
- ２段階ブロック化
  - l ビットごとにブロック化（大ブロック）
    - 先頭のrankはすべて記憶
    - 途中については、大ブロック先頭からの差分で記憶する
  - それぞれのブロックをwビットごとにブロック化
    - この中は先ほど（その１）と同様の計算を行う
  - すると、それだけで
    - $\binom{n}{l}\log n + \binom{n}{w}\log l + 2^w \log w$ bitで格納できる
    - これは$l = \log^2 n$、$w = (\log n)/2$ の時、$o(n)$ビット　→　無視できる！

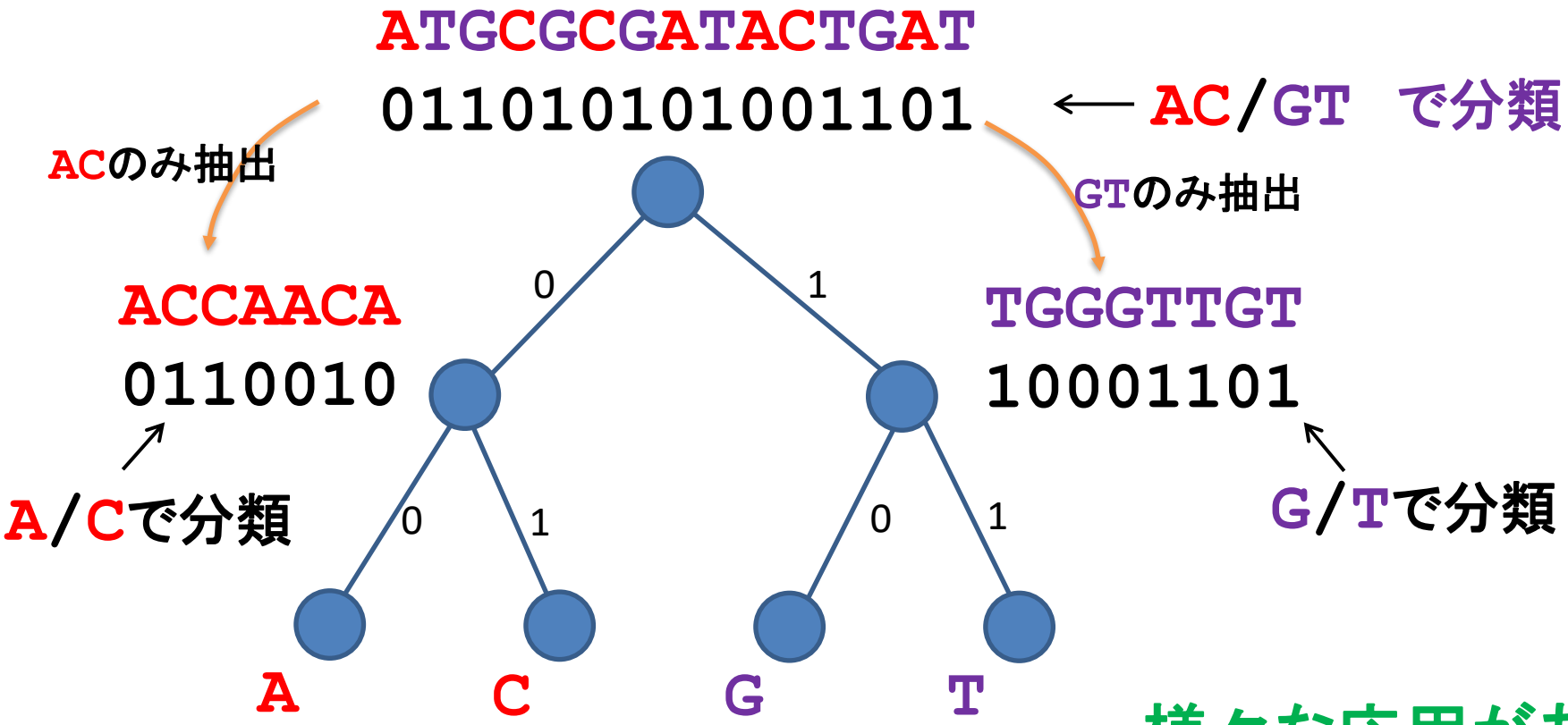


- select も同様のことが可能だが、通常は2分探索で実現することが多い
- アルファベットサイズが大きい場合も同様のことは可能だが、大きくなってしまう。

Wavelet 木

Wavelet 木（概要）

- 文字列に対するrank/selectをアルファベットに関する2分探索的に行うデータ構造
  - それぞれの01列に対し先ほどのデータ構造を作成すれば、bit列の場合のlog(アルファベットサイズ)倍＝文字列サイズですむ



様々な応用がある

Yet Another 圧縮索引： 圧縮接尾辞配列

- 接尾辞配列を（もう少しストレートに）圧縮するには？
  - Ψ（プサイ、接尾辞配列上のSuffix Link）を用いることが可能
    - すなわち、FM-Indexとは完全に逆の考え方！
  - エントロピーに比例した圧縮が可能
  - 順方向の検索がメイン となる

```
 0: mississippi$
 1: ississippi$
 2: ssissippi$
 3: sissippi$
 4: issippi$
 5: ssippi$
 6: sippi$
 7: ippi$
 8: ppi$
 9: pi$
10: i$
```

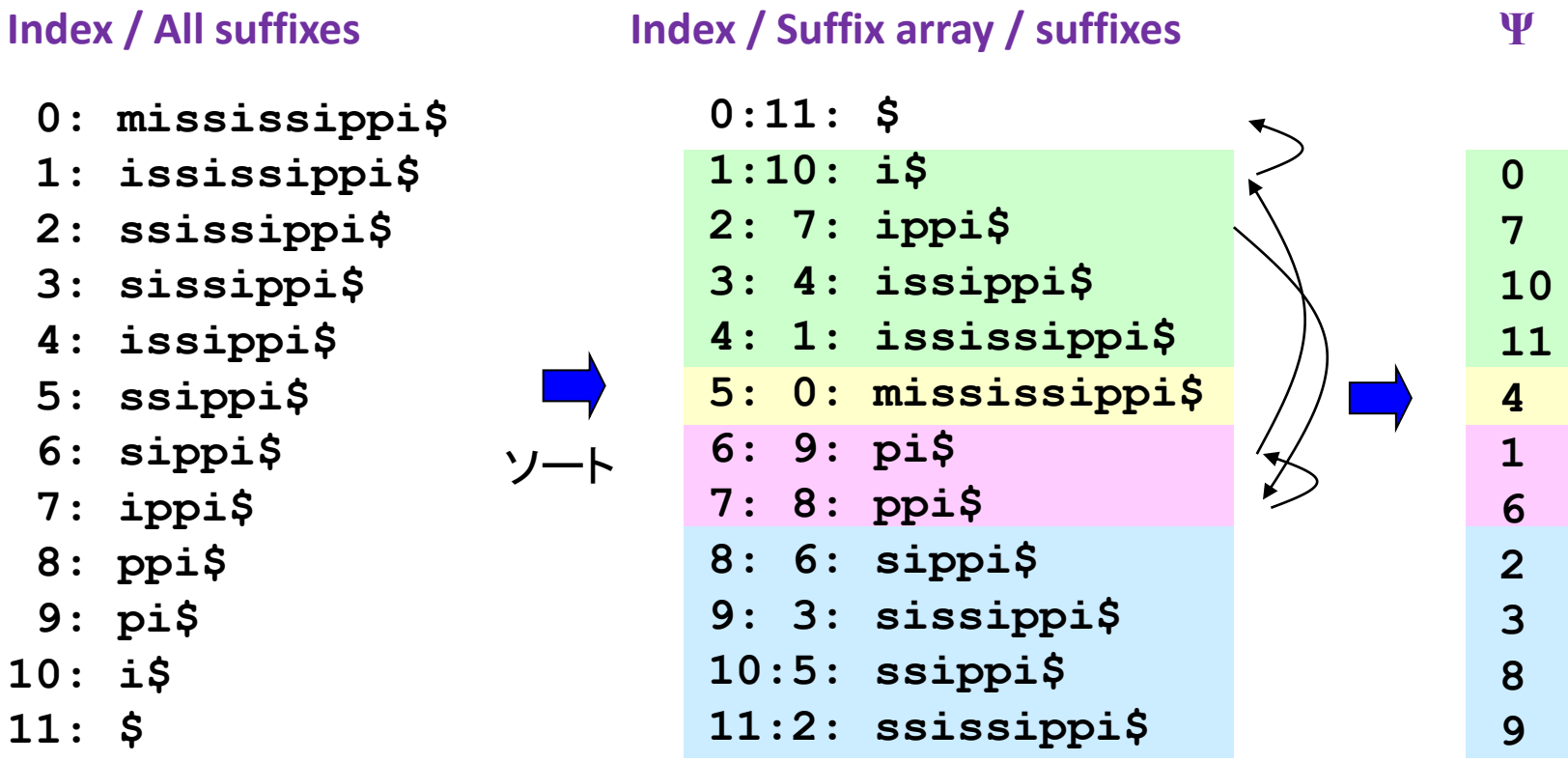
ソート

```
10: i$
 7: ippi$
 4: issippi$
 1: ississippi$
 0: mississippi$
 9: pi$
 8: ppi$
 6: sippi$
 3: sissippi$
 5: ssippi$
 2: ssissippi$
```

接尾辞配列

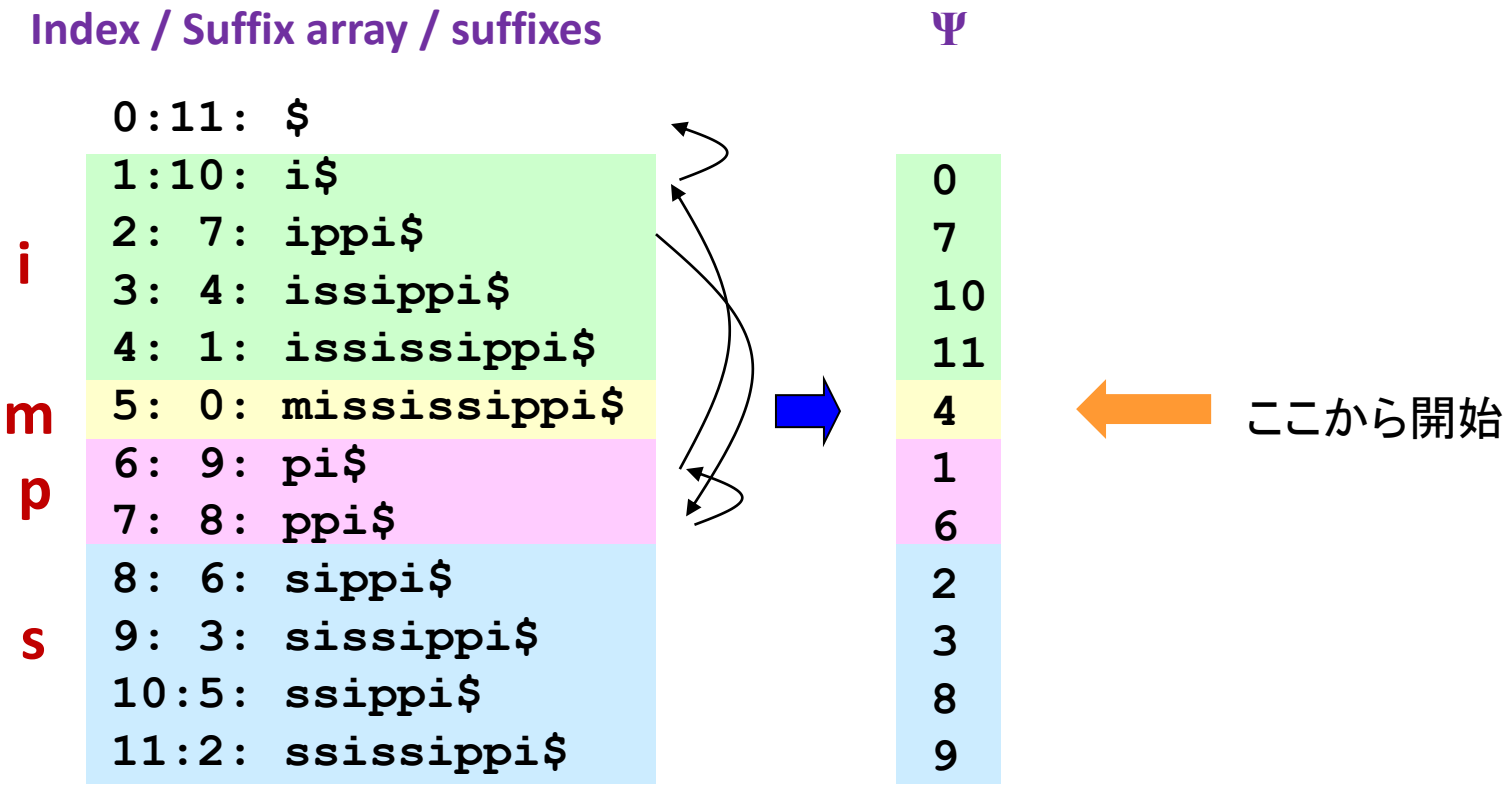
Suffix Array 上の Suffix Link

- Ψ（プサイ）（接尾辞配列上のSuffix Link）
  - 逆接尾辞配列から簡単に作成可能
- Textは記憶しなくてよい
  - 必要ならばΨ（とアルファベットリスト）から計算可能なため



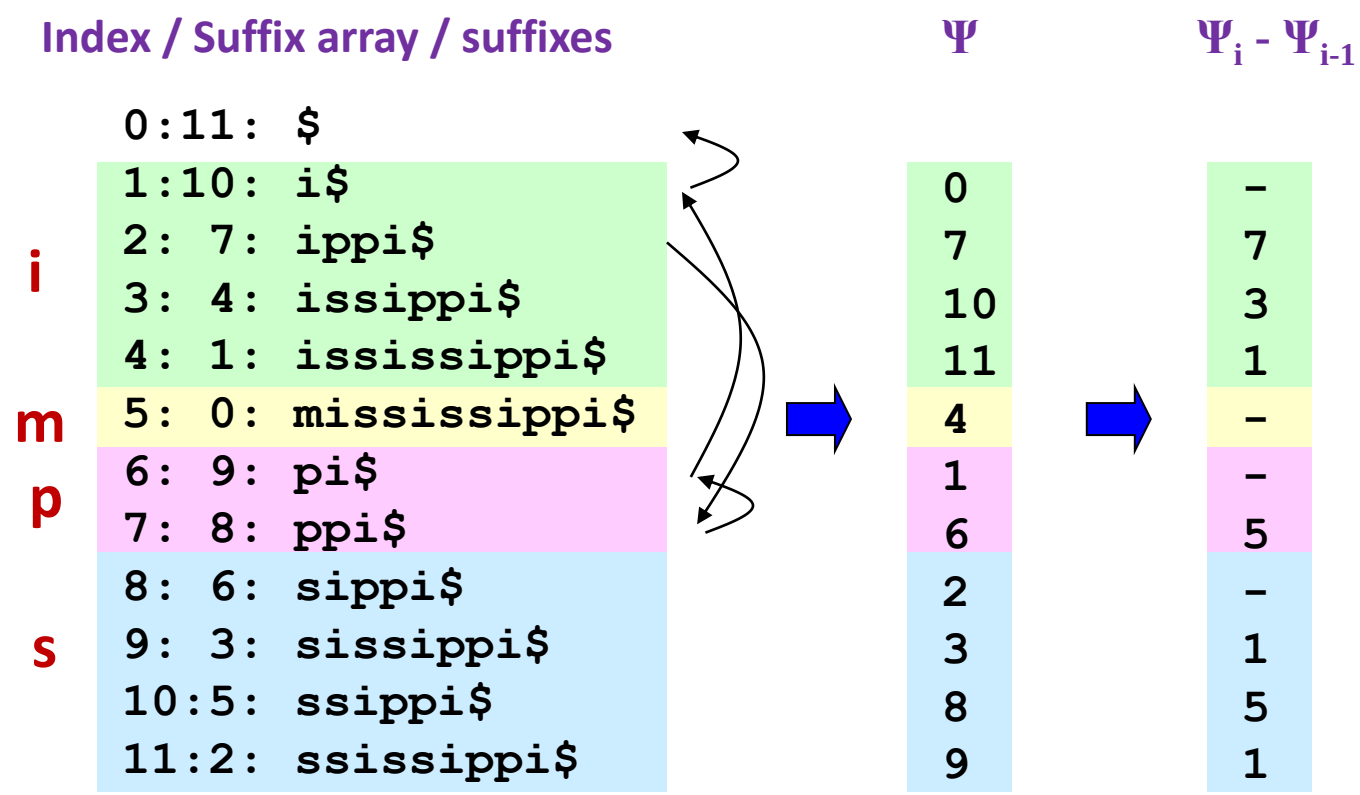
Ψからのテキストの復元

- 一番長い接尾辞がどこにあるかを覚えておいて、そこからΨを辿っていくだけで、復元できる
  - アルファベットがそれぞれ何個あるかは覚えておく



Ψの圧縮

- アルファベットごとに区切ると単調増加
  - 差分で覚える
    - すると、小さい値が非常に多くなり、圧縮しやすくなる
  - しかも、似たもののsuffix link先も似ているため類似部分列も多い
    → 圧縮効率も良い！
    - *cf.* 接尾辞木のDAG化



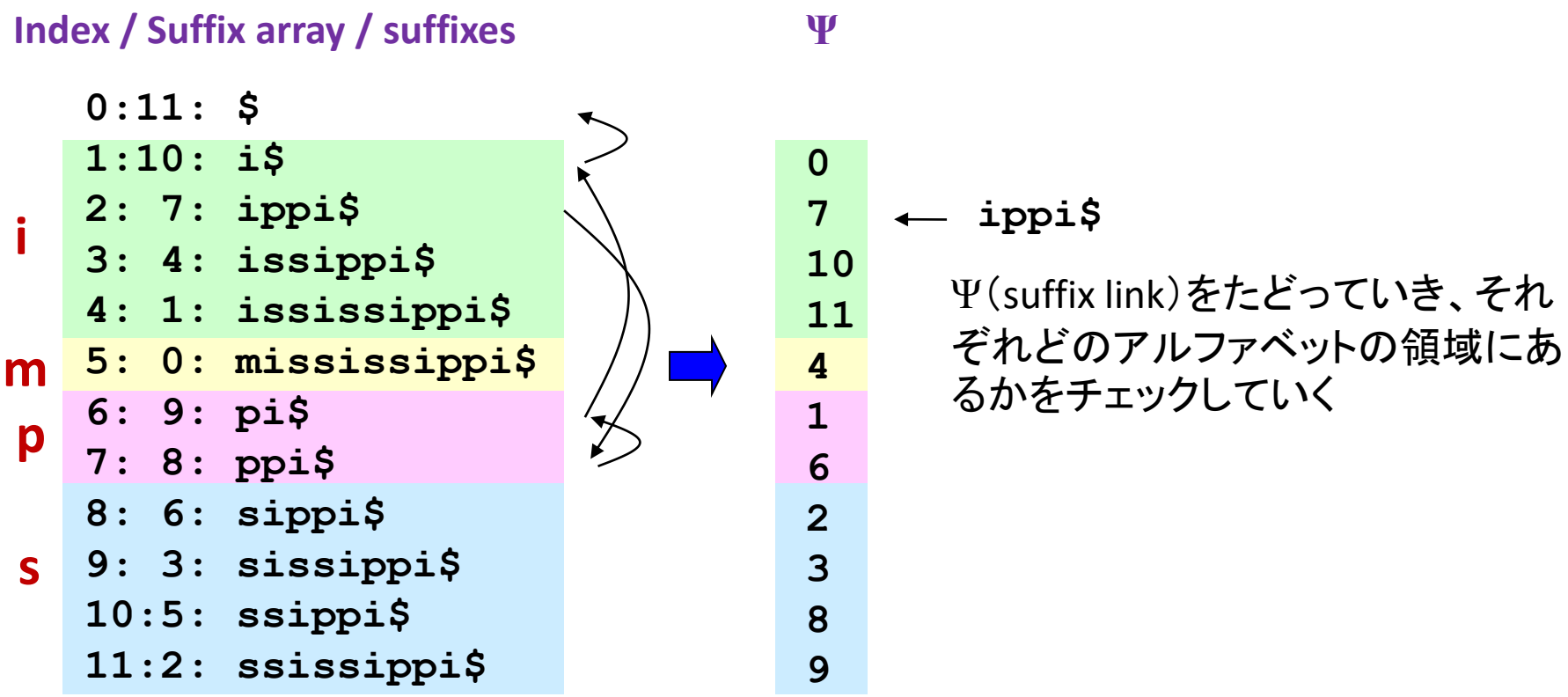
Ψの圧縮

- しかし、差分で覚えると、実際の値を計算するのに時間がかかってしまう
  - 間引いて途中の値を適当に覚えておけばよい
  - $O(n/\log n)$程度のサイズに間引けば、$O(\log n)$時間で計算できる、といったかんじ
    - スピード⇔圧縮率　のトレードオフあり

元の数列:	3	4	7	8	13	24	26	32	37	45	54	61	70	...
差分:		−	1	4	1	5	9	2	6	5	8	9	7	9 ...

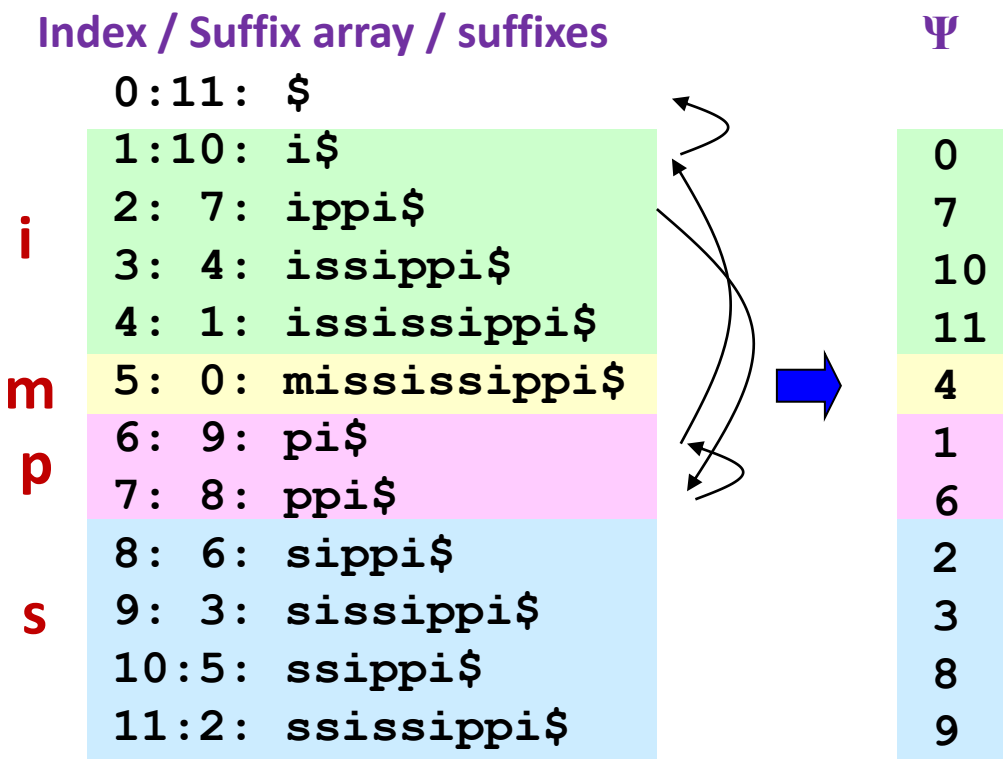
Ψを用いた検索

- SA上のある位置のsuffixはΨから簡単に復元可能
- これを利用して二分探索が普通にできる
  - $O(m \log n)$



検索結果の位置を知るには？

- SA[i]の値がわかればよい
  - テキスト全体の復元と同様にΨを辿っていけば、計算できる
  - ただそれだと$O(n)$時間かかってしまうため、間引いた接尾辞配列をもっておけば、その計算時間を短縮できる。
    - たとえば$O(n / \log n)$サイズにまで間引いておけば大幅に時間が短縮できる上に、さほどサイズは大きくなくてすむ。
    - こうしておけば、テキストを部分的に復元することもできるようになる。



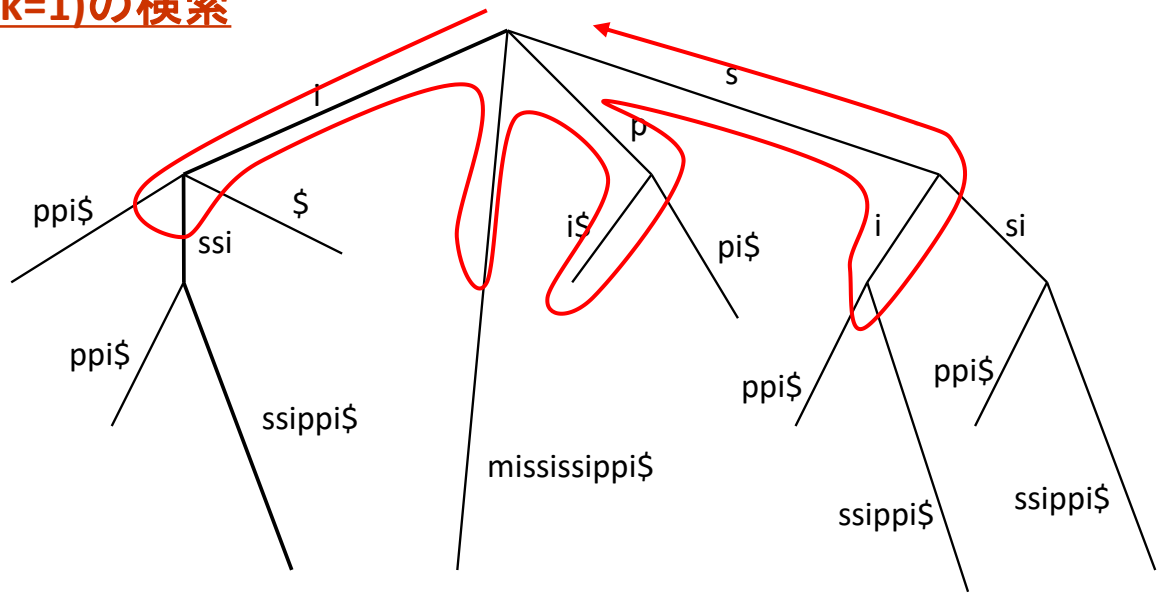
あいまい検索：Inexact Matching Problem

- 問題
  - テキストの中からクエリーとのedit distanceがk以下のsubstringを「すべて」見つける
    - テキストに対して前処理を行ってもよい
- 2種類の攻略法
  - 接尾辞木、接尾辞配列等を利用する
    - kに関して指数的な計算量となる
  - 小さなsubstringを用いてexact matchingによる検索を行うことでフィルタリングを行う
    - 実際にはその中間的な手法も存在

接尾辞木、接尾辞配列の利用

- 接尾辞木上で単に深さ優先探索でerrorが k を超えない地点まで探索する
  - あるいはそれを接尾辞配列（等）でシミュレート
  - 探索時には通常のDPによる比較を行う

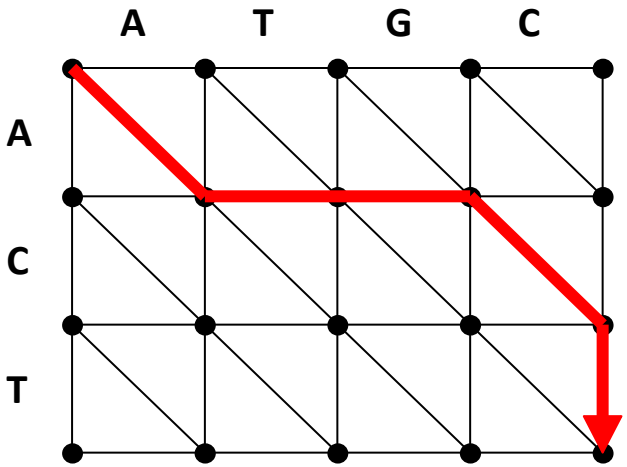
isi (k=1)の検索



Suffix tree of 'mississippi$'

接尾辞木上でのEdit Distance比較

- 動的計画法
  - 格子状のグラフで最長路を求める
  - 編集距離に上限kがあるので $O(km)$
    - m: クエリーの文字列の長さ
  - ノードごとにDP情報を保存する



ATGC-
A--CT