2010 6/14 情報論的学習理論と機械学習研究会 (IBISML) @東京大学

大規模文字列解析の理論と実践

岡野原　大輔
Preferred Infrastructure (PFI)

発表の流れ

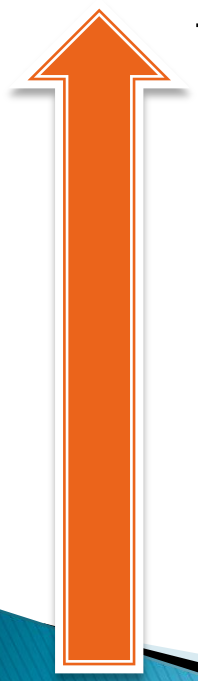
- 背景
- 文字列を扱うためのデータ構造
  - Wavelet Tree
  - 接尾辞配列 / FM-index / 接尾辞木
- 文書集合に対する高速な統計量の計算
  - 全部分文字列の頻度 / 重み付き全部分文字列の頻度
- 機械学習への応用
  - 全部分文字列を用いた文書分類
  - Sequence Memoizer
    - 無限長履歴 + 階層型Pitman-Yor過程

文字列データ

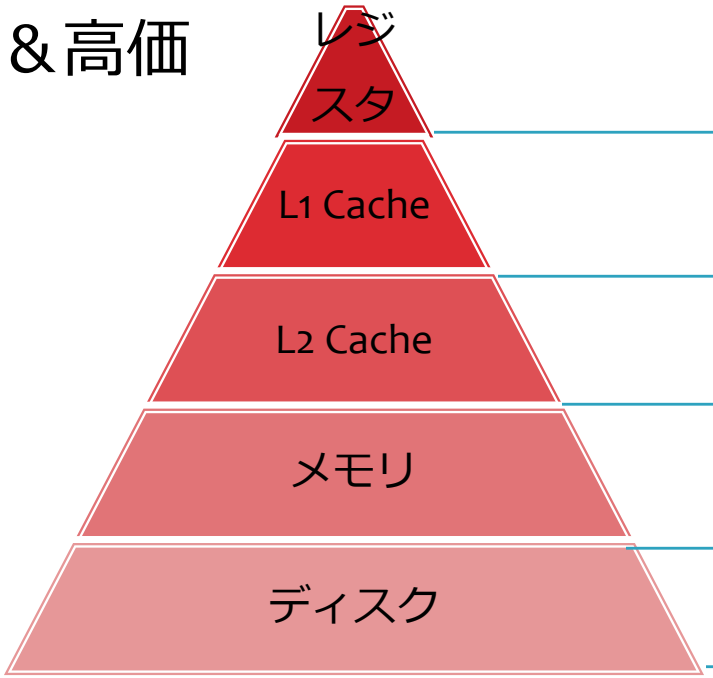
- あらゆる分野にみられる基本的なデータ形式
  - 自然言語、ゲノム、ログ
  - 対象量は急拡大：‘00年 10^6 単語 ⇒ ‘10年 10^{10}〜 10^{12}単語
- 文字列の性質は様々
  - 文字列長: 10^1(ログ)〜 10^9(ゲノム)
  - 文字種類数 : 4 (ゲノム) 〜 10^4 (日本語)
  - その他：繰り返し、頻度の偏り、マルコフ性
- ロバストなデータ構造とアルゴリズムが重要
  - 長さが数百億〜1兆、文字種類数が2〜数億、繰り返しや高頻度単語があっても安定して高速に動く

高速化の肝：省容量化

- 記憶階層間の速度差は年々大きくなっている
  - ディスクシーク1回 = 約3000万命令 (@3GHz CPU)
  - 上位の記憶階層での操作が最重要！



高速＆高価



	容量	参照時間 (nsec.)
レジスタ	10〜100word	0.3
L1 Cache	32KB	0.5
L2 Cache	256KB	7.0
メモリ	8GB	100
ディスク	1TB	10000000

2009年の標準的な構成　[J. Dean, LADIS 2009]

本発表のゴール
全部分文字列の統計量

- 入力：文書集合 $D=\{d_i\}_{i=1\ldots|D|}$
  - n: 全文書文書長
  - Σ: アルファベット集合，　$\sigma=|\Sigma|$
- $f(d_i,q)$: 部分文字列qのd_i中の出現回数
- 次の統計量を全ての部分文字列qについて求める

1. **出現頻度**　$f(q,D) = \sum_i f(d_i,q)$
2. **重み付き出現頻度**　$wf(q,D,\boldsymbol{r}) = \sum_i r_i f(d_i,q)$

- これらは、O(n) 時間, $n \log_2 \sigma$ bitで求められる
  - 長さ、頻度、ヒューリスティックスで打ち切らない
  - 日本語Wikipedia　($n=1.0\times10^9$　$\sigma=10000$)　250秒　2GB
  - ヒトゲノム　($n=3.0\times10^9$　$\sigma=4$)　750秒　750MB

準備：簡潔データ構造

- Rank辞書
- Wavelet Tree

Rank辞書

- ビット列 B[1, n]に対し次の操作を備える
  - $\mathrm{rank}_c(B, i)$: B[1…i] 中の$c \in \{0, 1\}$の数を返す

例： B=0110011100

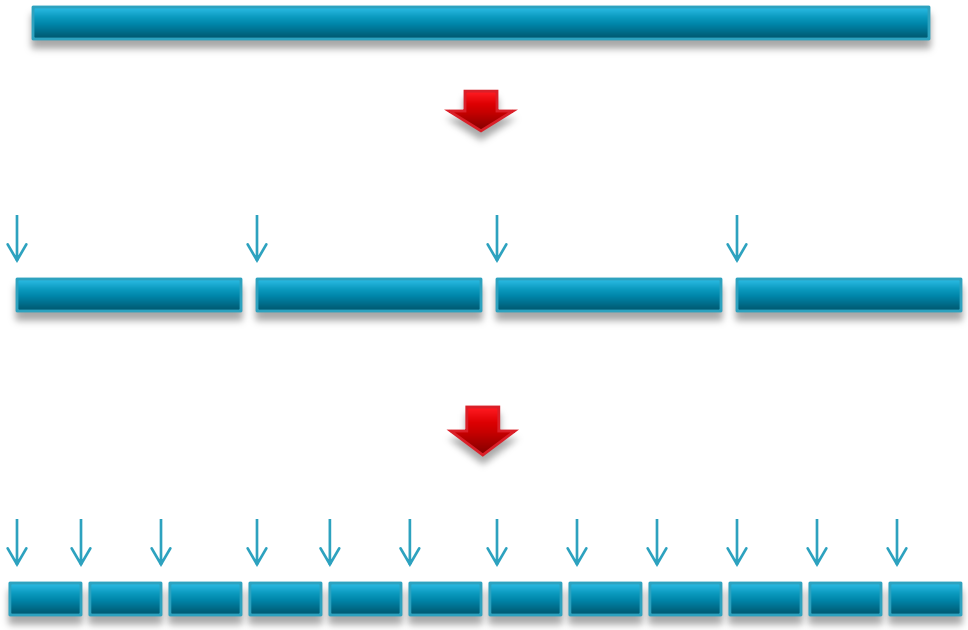
i	1	2	3	4	5	6	7	8	9	0
	0	1	1	0	0	1	1	1	0	0

$\mathrm{rank}_1(B, 7) = 4$

$\mathrm{rank}_0(B, 5) = 3$

- これをO(1)時間 n + o(n) bitsで実現したい

Rank辞書の実現



1. 長さ$L:=\log^2 n$のブロックに分割
   - 各ブロックの**rank**をLTに記録
   - 容量： $(n/L)\log n = o(n)$

2. 長さ$S:=(\log n)/2$の小ブロックに分割
   - 各小ブロックの**rank**をSTに記録
   - 対応大ブロックとの相対値
   - 容量： $(n/S)\log L = o(n)$

3. 長さSの全通りのビット列に対する**rank**の結果を記録
   - 容量：$2^S = o(n)$

$$\mathrm{rank}_1(B, i) = LT[i/L] + ST[i/S] + (\text{残りのrank})$$

計算量: $O(1)$

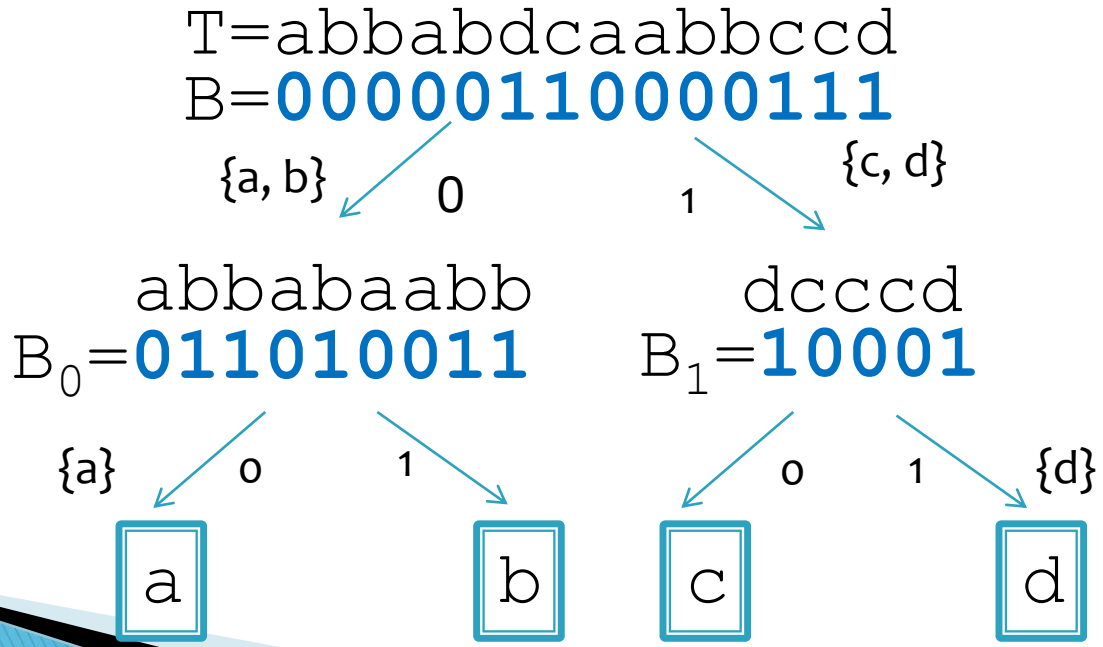
作業領域量： $n + o(n)$ bits

Wavelet Tree (WT) [Grossi+ SODA 03]

- 文字種類数$\sigma>2$の場合にrankを拡張する
- 入力: $T[1...n]$, $T[i] \in \Sigma$, $\sigma=|\Sigma|$
- $rank_c(T, i)$: $T[1...i]$ 中の$c \in \Sigma$の数を返す
  - ビット列と同じアプローチでは容量が大きくなる
- WTはこの問題を複数のrank辞書の操作に分解
  - $O(\log \sigma)$時間、$n\log_2\sigma$ bit (1文字あたり$\log_2\sigma$)

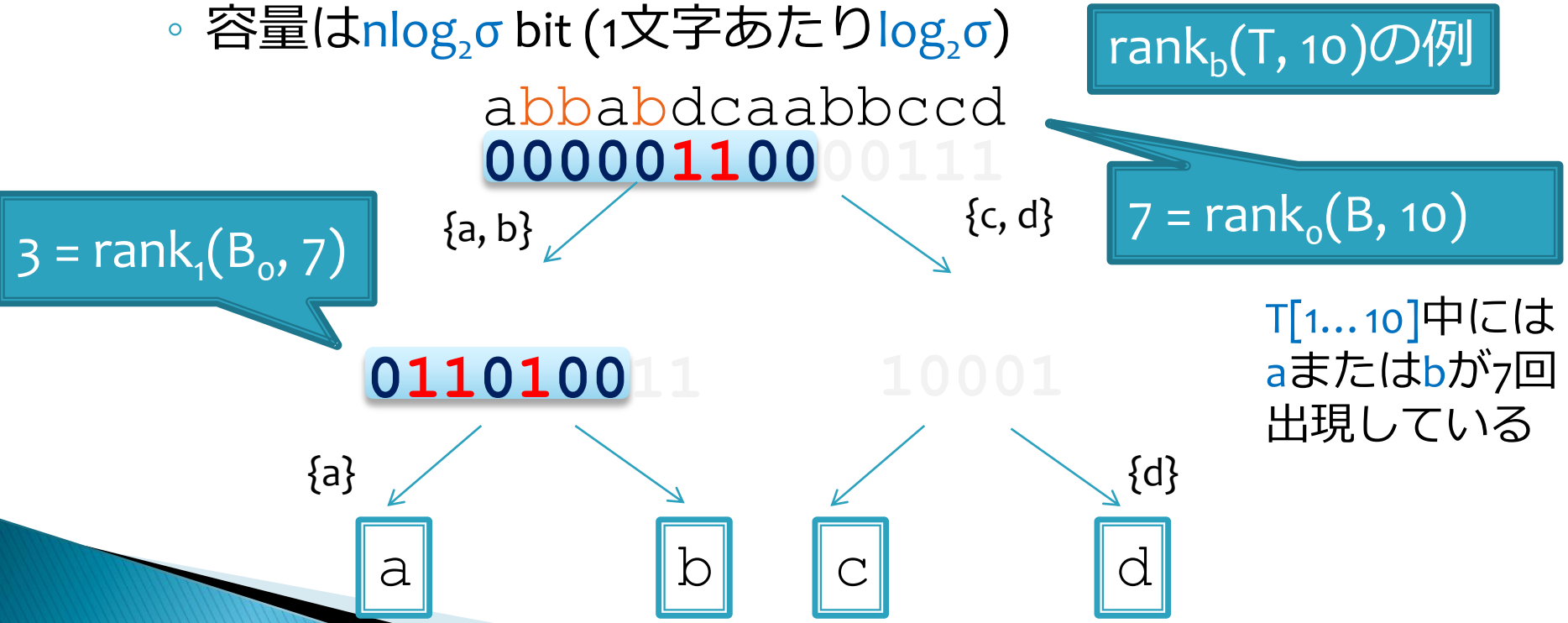
Wavelet Tree

- アルファベット集合を再帰的に分割
  - 分割後の順序は元の順序のまま
- 各ノードはビット配列を保持
  - 対応する文字が左へ行ったら0, 右へ行ったら1



Wavelet Tree

- $rank_c(T, i)$ は根からcに対応する葉に向かって各節点上で再帰的にrankを実行
- 1回のrankは$O(1)$、全体 $O(\log \sigma)$時間
  - 容量は$n\log_2\sigma$ bit (1文字あたり$\log_2\sigma$)



文字列を扱うためのデータ構造

- 接尾辞配列
  FM-index
  接尾辞木

全文索引

- 文字列に対して効率的に処理を行うために
  索引を構築する
  - 対象テキストを毎回全て調べるのは計算量が大きすぎる

全文索引の比較

	構築時間	索引容量	出現数計算 t_{occ}	出現位置計算	全部分文字列の統計量
FM-index	O(n)	n log σ	O(|q| log σ)		×
SA	O(n)	n(log n +log σ)	O(|q| log n)	t_{occ} + O(f(q))	×
ST	O(n)	O(n log n)	O(|q|)	t_{occ} + O(f(q))	○

n = 文字列長 σ = アルファベット数

文字列を扱うデータ構造の比較
実際のパフォーマンス

	構築時間 (sec.)	索引容量 (入力比)	出現数計算 (sec.)	出現位置計算 (sec.)	全部分文字列の統計量
FM-index	250	1	2.5×10^{-6}		×
SA	250	5	5.0×10^{-6}	5.0×10^{-6}	×
ST	872	10〜30	2.5×10^{-6}	2.5×10^{-6}	○

- 1GBの入力 ($N = 2^{30}$, $\sigma = 256$)
- CPU 3GHzの場合の例

- 索引容量は元文書が必要な場合はそれを含む
- 計算時間の殆どはキャッシュミスが占める
  - 出現位置の計算時はキャッシュミスが1回しか起きない

接尾辞配列 (SA) [U. Manber 91]

入力 T=abracadabra$ （$はT中に出現しない最小の文字）

(1) 接尾辞S_i=T[i…n]を列挙

S_1 abracadabra$
S_2 bracadabra$
S_3 racadabra$
S_4 acadabra$
S_5 cadabra$
S_6 adabra$
S_7 dabra$
S_8 abra$
S_9 bra$
S_{10} ra$
S_{11} a$
S_{12} $

(2) 辞書式順序でソート

S_{12} $
S_{11} a$
S_8 abra$
S_1 abracadabra$
S_4 acadabra$
S_6 adabra$
S_9 bra$
S_2 bracadabra$
S_5 cadabra$
S_7 dabra$
S_{10} ra$
S_3 racadabra$

(3) 添字番号をSA[1,n]に記録

SA
12
11
8
1
4
6
9
2
5
7
10
3

SAによる検索

これらの接尾辞は明示的に保存せず
T[SA[i]…]で参照

abraを検索

SA上で二分探索を行う

abraとadabra$を比較

SA	Suffix
12	$
11	a$
8	abra$
1	abracadabra$
4	acadabra$
6	adabra$
9	bra$
2	bracadabra$
5	cadabra$
7	dabra$
10	ra$
3	racadabra$

SAによる検索

q = abraを検索

abraとabra$を比較

SA	Suffix
12	$
11	a$
8	abra$
1	abracadabra$
4	acadabra$
6	adabra$
9	bra$
2	bracadabra$
5	cadabra$
7	dabra$
10	ra$
3	racadabra$

SAによる検索

abraで始まる接尾辞の領域が分かる

sp

ep

sp : 開始位置
ep : 終了位置

abraはSA[sp,ep]= {8, 1}で出現している

出現回数はep – sp + 1=2

SA	Suffix
12	$
11	a$
8	abra$
1	abracadabra$
4	acadabra$
6	adabra$
9	bra$
2	bracadabra$
5	cadabra$
7	dabra$
10	ra$
3	racadabra$

SAの構築

- 構築はO(n) 時間, $n(\log_2 n + \log_2 \sigma)$ bits
  - SAIS : σに依存せず、安定して高速 [G. Nong+ DCC 09]
  - http://sites.google.com/site/yuta256/sais
- 長さtの文字列の先頭への追加はO(t log n)時間
  - 後述のBWTを介して可能

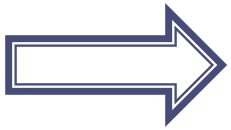
Burrows Wheeler 変換 (BWT)

[M. Burrows+ 94]

- 文字列Tから文字列Bへの可逆変換
- B[i] := T[SA[i]-1]　(B[i] = T[n] for SA[i]=0)

各接尾辞の直前の文字

a	$
r	a$
d	abra$
$	abracadabra$
r	acadabra$
c	adabra$
a	bra$
a	bracadabra$
a	cadabra$
a	dabra$
b	ra$
b	racadabra$



B = ard$rcaaaabb

Block Sorting という名前で発表された

BWTの構築

- SAを構築し定義に沿ってBWTを構築
  - O(n) 時間　$n \log_2 n$ bit
- SAを経由せず BWTを直接構築 [Okanohara+ SPIRE 09]
  - O(n) 時間　$O(n \log \sigma)$ bits for $\sigma = (\log n)^{\Omega(1)}$
- 動的構築
  - $O(n \log n)$ 時間, $n \log \sigma$ bits

```
Input T[1..n]
p = 0
for i = n to 1
    B[p]にcを挿入
    p = CF[c] + rank_c(B, p)  (CF[c] c未満の文字の出現回数)
```

BWTの特徴

- 可逆変換
  - BのみからTを復元できる
- 圧縮しやすい　c.f. bzip2
  - 同じ文字が並びやすい
  - 圧縮率はk次経験エントロピーに漸近 [H. Kaplan+ TCS 07]
- 接尾辞配列と同等の情報を備える
  - Bのみを用いて全文検索が可能
  - 元の文書Tは必要ない
    - self-index : 索引のみから元文書の情報を参照可能

BWTの性質 (1/3)

- F[i] := T[SA[i]]を考える
  - 各接尾辞の先頭文字を並べた配列
  - T中の文字を小さい順に並べる

B	F
a	$
r	a$
d	abra$
$	abracadabra$
r	acadabra$
c	adabra$
a	bra$
a	bracadabra$
a	cadabra$
a	dabra$
b	ra$
b	racadabra$

BWTの性質 (2/3)

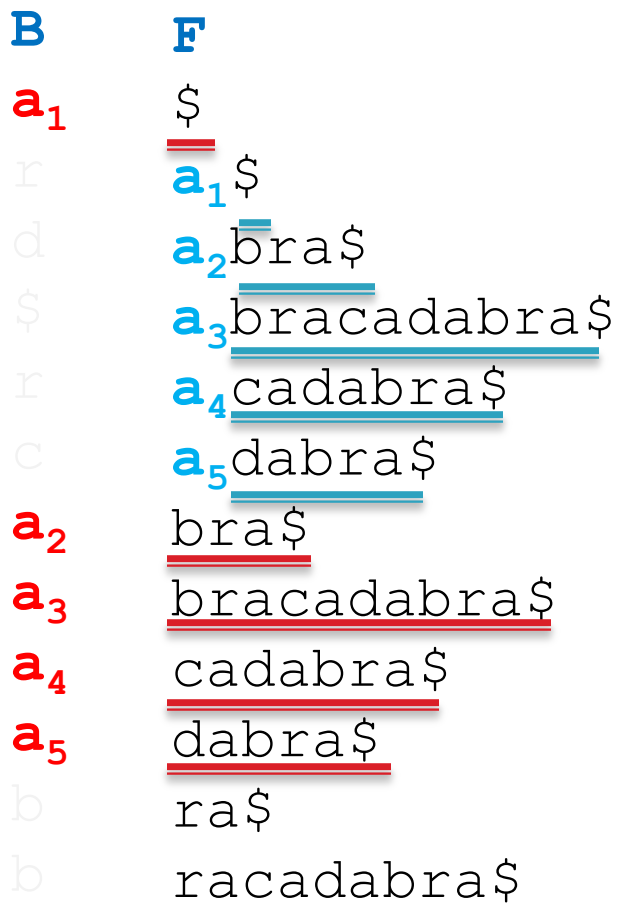
- B中に出現している文字はFではどこに出現しているか？

B	F
a_1	$
r	a$
d	abra$
$	abracadabra$
r	acadabra$
c	adabra$
a_2	bra$
a_3	bracadabra$
a_4	cadabra$
a_5	dabra$
b	ra$
b	racadabra$

BWTの性質 (3/3)

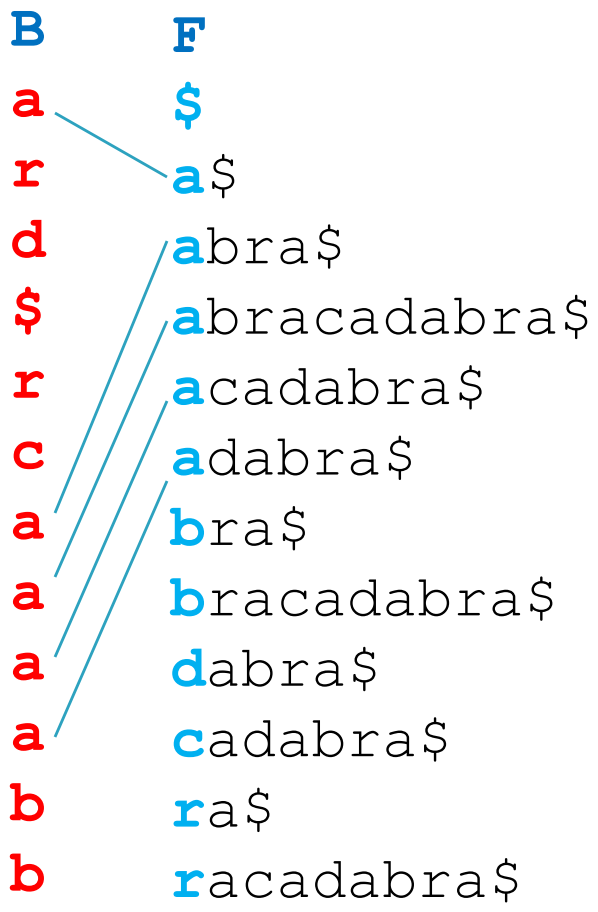
- BとFの各文字の順序は後続の文字列で決定

- 同じ文字の順序について注目
  - Bの順序決定に使う文字と
    Fの順序決定に使う文字は同じ
  - F中の順序は1文字目が同じ文字
    なので2文字目以降で比較
- 同じ文字はBとFでは同じ
  順序で出現する



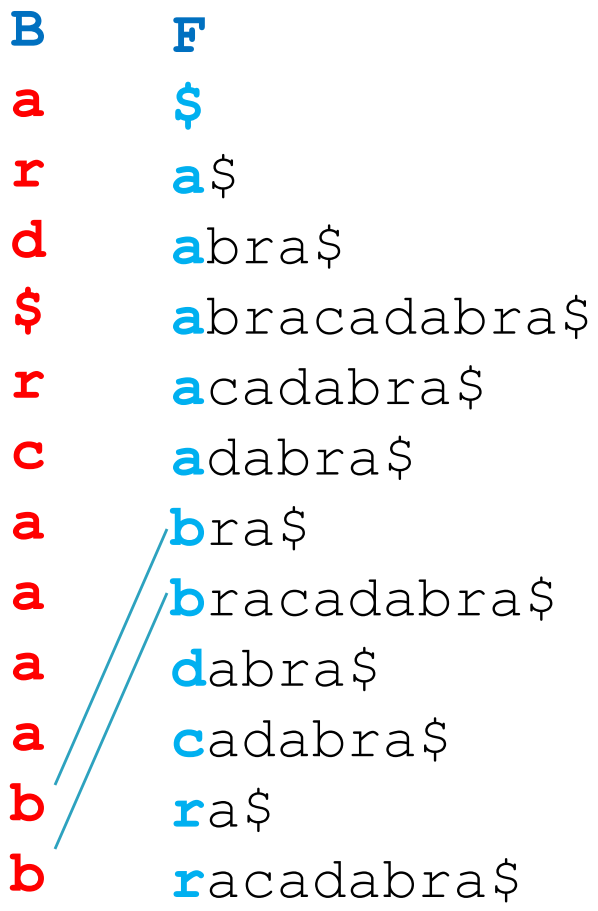
例：aの場合

- 各$c \in \sum$についてFとBでは同じ順序で出現



例：bの場合

- 各c∈∑についてFとBでは同じ順序で出現

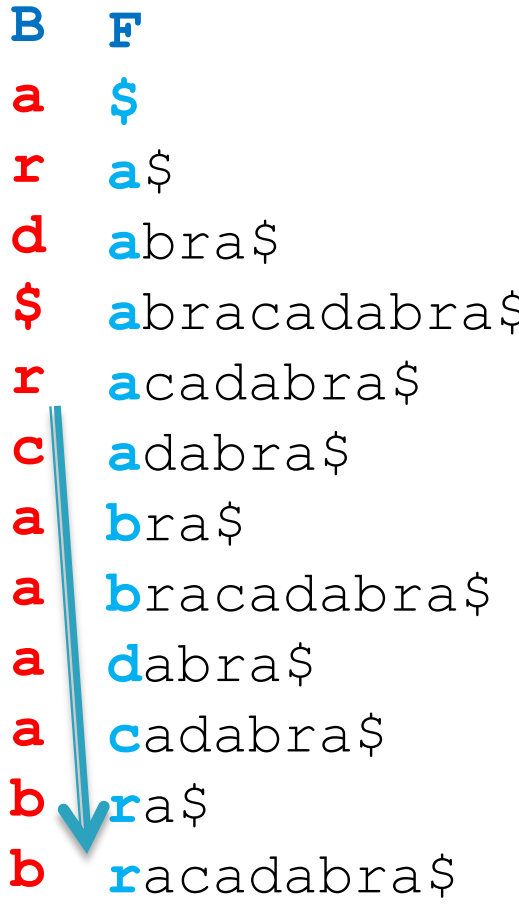


例：rの場合

- 各$c \in \sum$についてFとBでは同じ順序で出現

LF-mapping [Burrows+ 94]

- BとFはどのように関係してるか？
- LF(i) := CF[c] + $\mathrm{rank}_c(B, i)$
  - B[i]=cの時、それに対応するFの位置
  - CF[c] = F中のc未満の文字の出現回数
- LFは一文字長い接尾辞の順位を返す
  - LF(i) = $SA^{-1}[SA[i]-1]$　SA^{-1}はSAの逆関数
  - 元のTで言えば、1文字前を返す

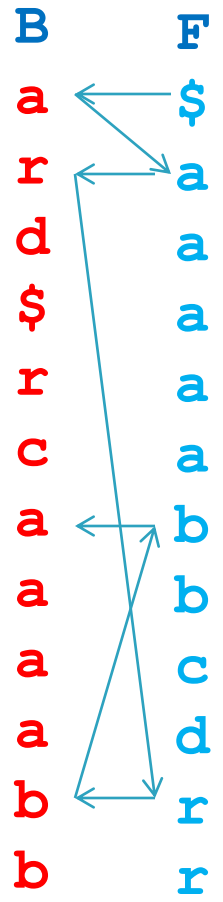


LF-mappingによるBからTの逆変換

- LF(i)はTでは1文字前を返す
  - 末尾から順に1文字ずつ復元する

```
1. Input: B
2. p = 1
3. for i = 1 to n
       T[n-i] = B[p]
4.     p      = LF(i)
5. end for
6. Output: T
```

T = …abra$



FM-index

[P. Ferragina+ FOCS 00]

- BWTのみを用いて全文検索が可能
  - LF-mappingと同じ考えが使える
- クエリq[1, m]を後ろから一文字ずつみていく
  - iステップ目では[sp, ep]はα=q[m-i, m]のSAの範囲を保持
  - 次の文字が c = q[m-1-i]の時cαの範囲(sp', ep')は？
  - $sp' = CF[c] + rank_c(B, sp-1)+1$
  - $ep' = CF[c] + rank_c(B, ep)$
- O(mlog σ) 時間 $n\log_2\sigma$ bit
  - BWTに対するWavelet Treeのみがあればよい

FM-indexの例

sp	e_{412}	arn many money …
	c	**at** is very cute.
	e_{413}	**at** many apples las
	n	**at** is the process …
ep	e_{414}	**at**s two hamburgers…
	e_{415}	conomy is not good…

atの範囲が分かっている時、eatの範囲を求めるには？

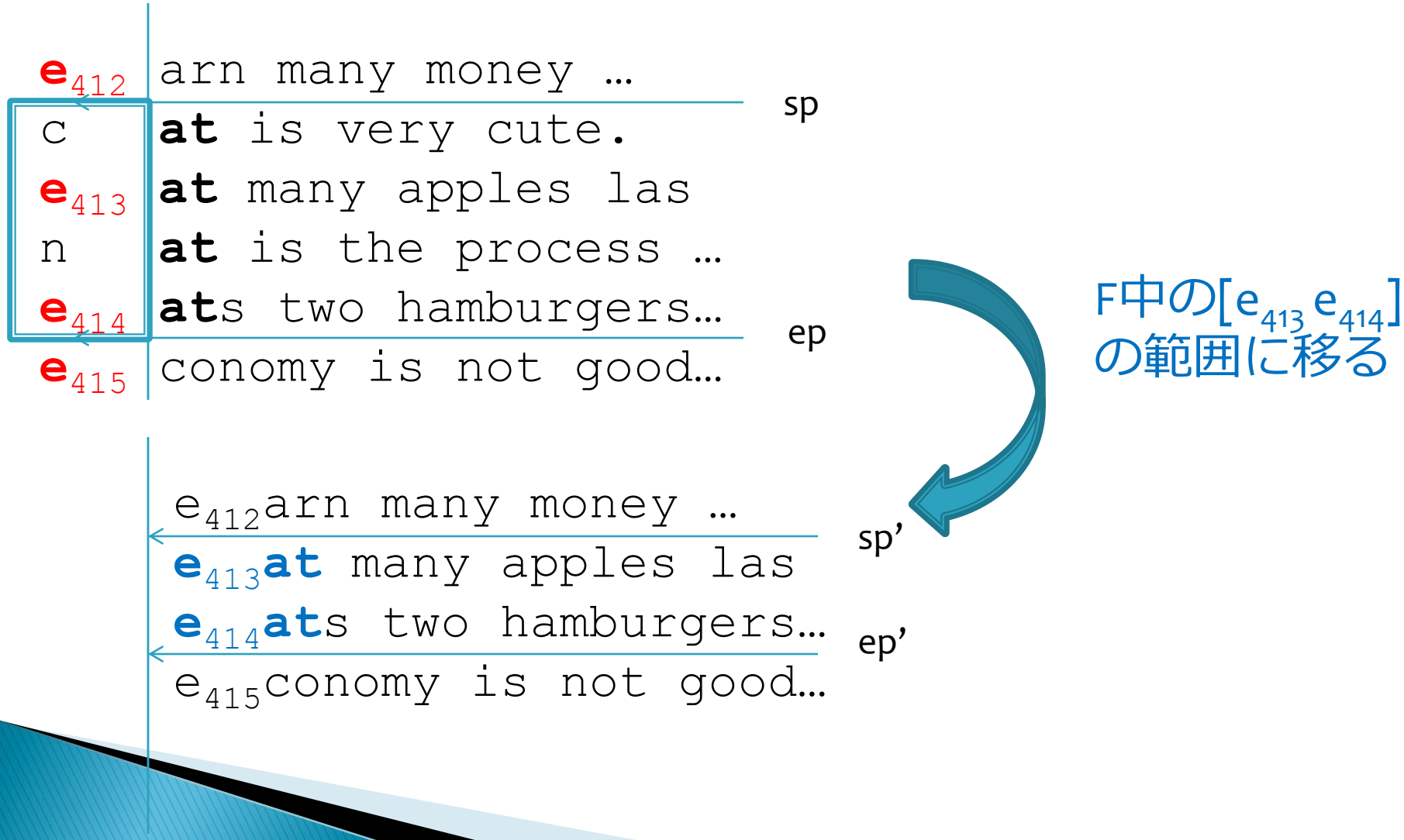
[sp,ep]の範囲中にあるeの番号の最初、最後を求める

左の場合e_{413}と e_{414}

これらの番号は

$rank_e(B, sp)$, $rank_e(B, ep)$ で求まる

FM-indexの例


e_{412} arn many money …
c at is very cute.
e_{413} at many apples las
n at is the process …
e_{414} ats two hamburgers…
e_{415} conomy is not good…
sp
ep
e_{412}arn many money …
e_{413}at many apples las
e_{414}ats two hamburgers…
e_{415}conomy is not good…
sp’
ep’
F中の[e_{413} e_{414}]
の範囲に移る

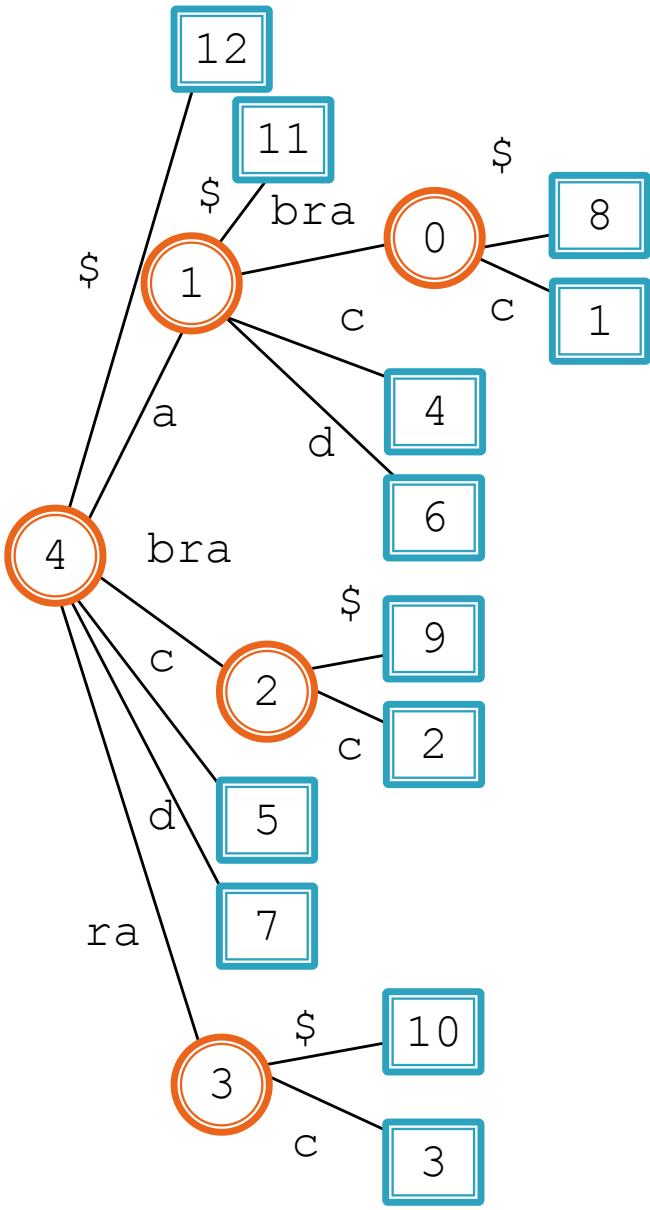
FM-index

```
input q[1, m]
i=m sp=1 ep=n
while (sp ≦ ep) & (i >= 0) do
    c  = q[i]
    sp = CF[c] + rank_c(B, sp-1)+1
    ep = CF[c] + rank_c(B, ep)
end while

output [sp, ep] //出現個数はep-sp+1
```

接尾辞木

- 入力T中の全ての接尾辞からなるtrieに対し子が一つの節点を取り除いたもの
  - 葉には出現位置を記録（SA）
- 部分文字列の探索はO(m)時間
  - 枝の文字列に従って根から辿る
  - 止まったところの子孫が出現箇所
- 性質：内部節点の数は高々n-1
  - 接尾辞を一つずつ追加して接尾辞木を作る場合、各操作で内部節点は高々一つしか増えない



接尾辞木のコンパクトな利用

- 接尾辞木はサイズが大きいのが問題
  - 通常 20N～40Nバイト
  - 親、子を指すポインタが大きい
- 節点さえ列挙できればよい
- SAのみから節点を列挙できる [Kasai+ CPM 01]
  - O(n) 時間, $3n\log_2 n$ byte
  - code.google.com/p/esaxx （サンプルコード）
- BWTのみから節点を列挙できる（次項）
  - O(n logσ)時間 $n \log_2 \sigma$ byte

BWTを利用した節点の列挙（概要）

[Okanohara to appear]

- FM-indexを利用してSTの深さ優先探索を行う
- Tを逆順にしたT^Rに対するBWT Bを構築する

funciton enumTree(sp, ep)　// [sp,ep]以下にある部分木を列挙
- B[sp, ep]中に出現する各文字の出現回数を求める*1
- 出現した各文字 c について
  - sp′ := CF[c] + rank_c(B, sp-1)+1
  - ep′ := CF[c] + rank_c(B, ep)
  - enumTree(sp′, ep′)

end function

- *1はwavelet treeを用いて出現する文字種類数がvの時 O(v log σ)時間で求められる
- もう少し工夫すると、全体でO(nlogσ) 時間で求められる

文書集合の統計量計算

- 出現頻度
- 文書頻度
- 重み付き頻度

文書集合の統計量

- 文書集合 $D=\{d_i\}_{i=1\ldots|D|}$
  - 全文書長をNとする
  - $f(d_i,q)$：qのd_i中の出現回数
- 次の統計量を全ての$q \in \sum^{\infty}$について求めたい
- 出現頻度　$f(q,D) = \sum_i f(d_i,q)$
- 重み付き出現頻度　$wf(q,D,\boldsymbol{r}) = \sum_i r_i f(d_i,q)$

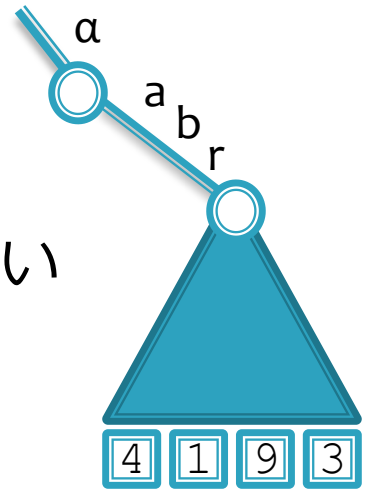
出現頻度 $f(q,D)=\sum_i f(d_i,q)$

- 文字列 $T:= d_1\#_1 d_2\#_2 \cdots d_{|D|-1}\#_{|D|-1} d_{|D|}$ を考える
  - 文書をつなげた文字列
  - $\#_i$ は文書中に出現しない特殊文字
- Tに対する接尾辞木 STを構築する
  - 節点が列挙できれば何でも良い
- ST中の節点を順に列挙し、それらの頻度を報告
  - それぞれの頻度はO(1)時間で報告できる
- 節点に対応しない部分文字列の情報は？

⇒節点の情報で全ての部分文字列の情報を含む

出現頻度 $f(q, D) = \sum_i f(d_i, q)$

- 同じ枝に属する部分文字列の出現頻度は同じ
  - 右の例でαa, αab, αabrの文字列の出現箇所はどれも4, 1, 9, 3
  - よって節点に対応するαabrの頻度のみを記録しておけばよい



- ST中の内部節点の数は高々N-1
- 全ての部分文字列の出現頻度情報はO(N)時間で報告でき、それらは2N-1のグループでまとめられる
  - N個の葉とN-1以下の内部節点に対応する部分文字列

重み付頻度 $wf(q, D, \boldsymbol{r}) = \sum_i r_i f(d_i, q)$

- 各文書d_iに対し重み$r_i \in R$が与えられる
  - 頻度を数える際、それが所属する文書の重みだけ足す
  - 機械学習・データマイニングで使われる
  - 例：正例の文書 $r_i = +1$　負例の文書 $r_i = -1$
- 同じ枝に属する部分文字列の統計量は同じ
  - $f(d_i, q)$の値が全て同じため

重み付頻度 $wf(q, D, \boldsymbol{r}) = \sum_i r_i f(d_i, q)$

- D[i]をSA[i]が属する文書番号とする
- [sp, ep]中の重み付き頻度は$\sum_{k=sp..ep} r_{D[k]}$
- $A[i] := \sum_{k=1}^{i} r_{D[k]}$ とおく
  - A[1,N]はO(N)作業領域、O(N)時間で構築可能
- 文字列qが節点(sp, ep)に対応する時

$$wf(q, D, \boldsymbol{r}) = A[ep] - A[sp - 1]$$

重み付出現頻度の例

- T = abab#ab#aab#　$(r_1\ r_2\ r_3) = (-1, 1, 2)$の時
- q=abの重み付き頻度を求める
- abの範囲は[2, 5]なので、abのwfはA[5]-A[2-1] = 1
  - 2+1-1-1 -1

$$A[i] := \sum_{k=1}^{i} r_{D[k]}$$

i	A	D	Suffix
1	2	3	aab#
2	4	3	**ab**#
3	5	2	**ab**#aab#
4	4	1	**ab**ab#ab#aab#
5	3	1	**ab**b#ab#aab#
6	5	3	b#
7	6	2	b#aab#
		…	

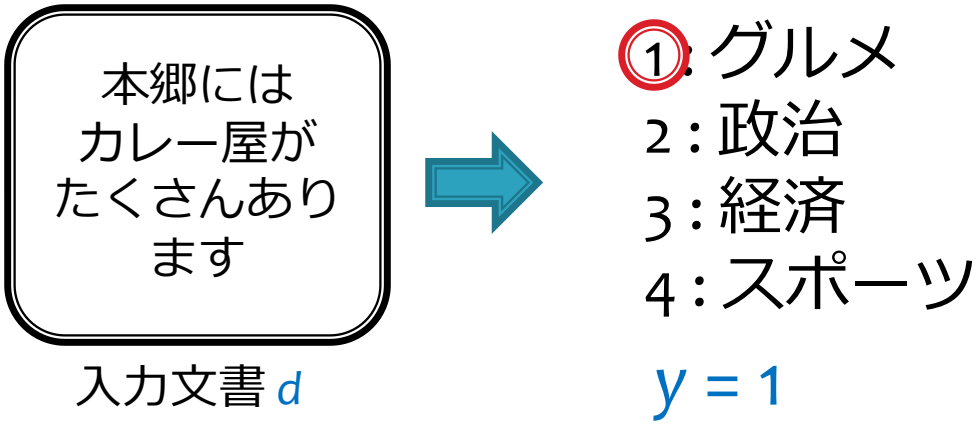
機械学習への応用

- 全部分文字列を利用した文書分類
  Sequence Memoizer

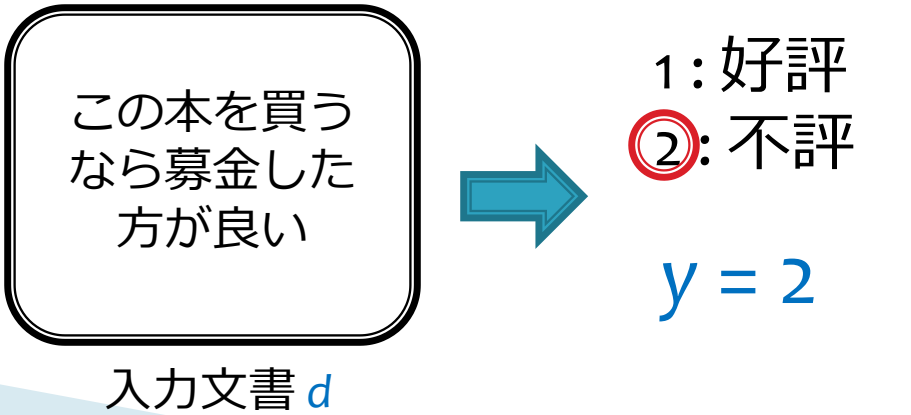
文書分類 問題

- 入力文書dのラベル$y \in \{1, \ldots, k\}$を求める

カテゴリ分類

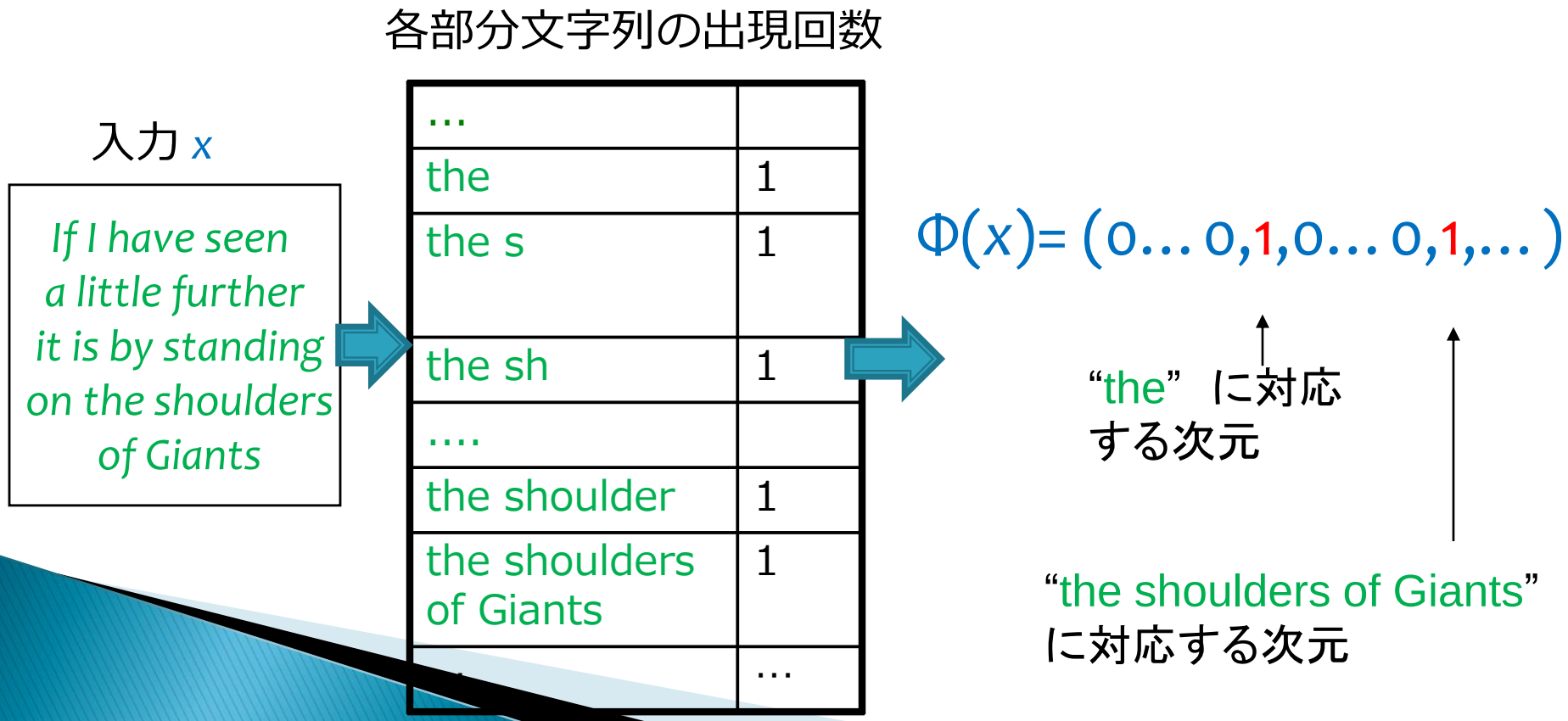


評判分類



全部分文字列特徴 [Okanohara+ SDM 09]

- 文書を全部分文字列からなるBag of Wordsで表現
  - 部分文字列長や特殊文字などで制限はしない



全部分文字列特徴を利用する従来手法との比較

N : 全文書長
D: 訓練文書数

	文字列カーネル [Vishwanathan+ 04]	SLR [G. Ifrim+ KDD 08]	提案手法 [Okanohara+ SDM 09]
手法	DPによるカーネルの高速結果	部分文字列をGreedyに探索	L_1 + Grafting + Suffix Tree
学習計算量	$O(N + D^2)$ または $O(ND)$	保証無し	$O(N)$
作業領域 (Byte)	20～40N	保証無し (約 10N)	2N ～5N
学習結果 (Byte)	約20N	約4N～10N	約N/100～N/10
特徴	•少数の特徴が効いている場合苦手 •結果がブラックボックス	•近似解 •過学習しやすい	•正則化有 •近似無で正確な結果 •数百万文書まで１台で処理可能

ロジスティック回帰モデル

$$p(y \mid x; \mathbf{w}) \frac{1}{Z(x)} \exp\left(\mathbf{w}^T \phi(x, y)\right)$$

入力ベクトルΦ(x)とラベルベクトルの直積

- 線形識別器を利用して分類
  - $\mathbf{w} \in R^m$は重みベクトル: 各特徴の重みを表す
  - カーネルを使わず特徴ベクトルを陽に表す
- 訓練はL_1正則化付最尤推定で行う
- 推定は $y^* = \mathrm{argmax}_y\, p(y|x; \mathbf{w})$

勾配の計算 = 重み付き頻度問題

- 対数尤度の勾配を求められれば良い

  $\mathbf{v} := \partial L(\mathbf{w})/\partial \mathbf{w}$

  $= \sum_{I, y} I(y_{D[i]} = y) - P(y|x_{D[i]}; \mathbf{w})\, \Phi(x_i, y)$
- 特徴kが文字列q、ラベルyに対応する場合
  - 文書の重みを $r_i := I(y_{D[i]} = y) - P(y|x_{D[i]}; \mathbf{w})$ とした時
  - $v_k = wf(q, D, \mathbf{r})$

⇒ 重み付き頻度問題に帰着

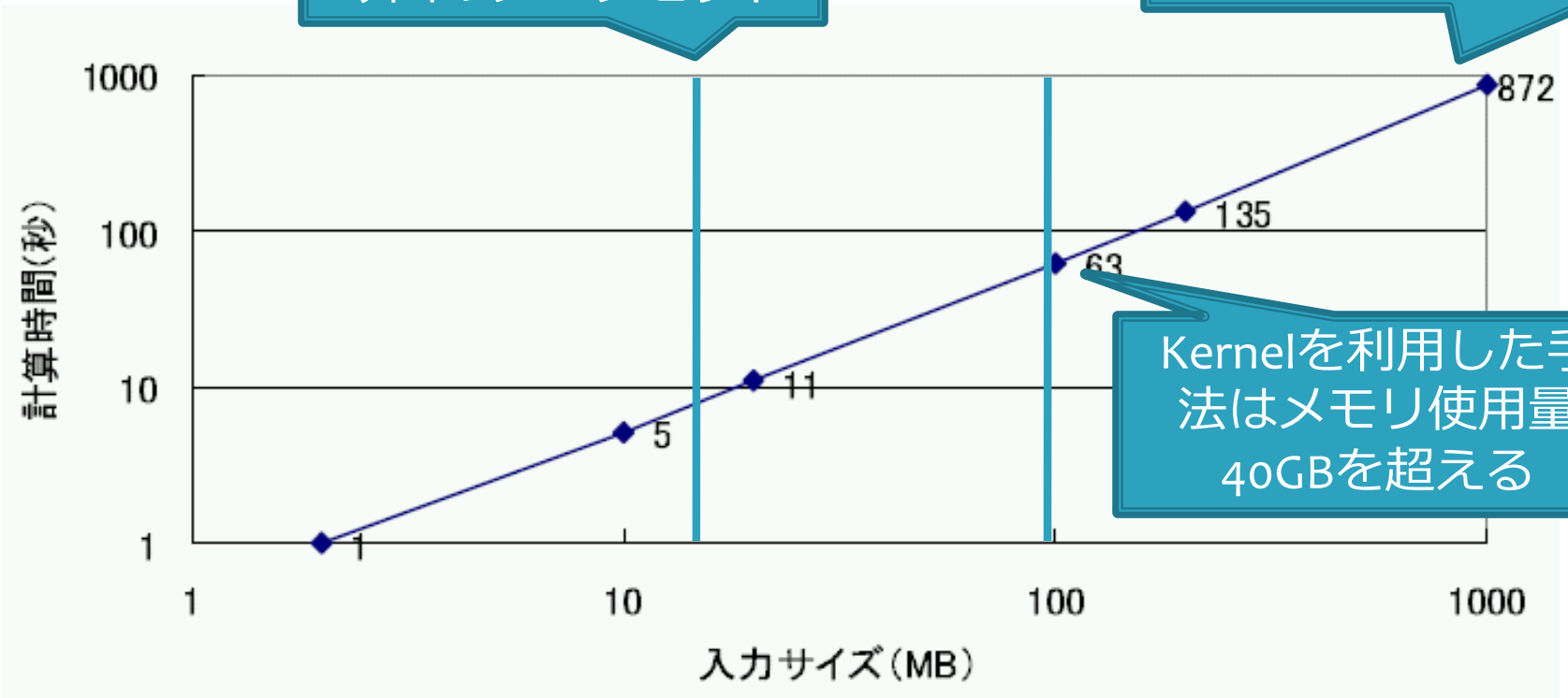
- O(n)時間で**v**を求めることができる
  - Grafting アルゴリズムと組み合わせることで有効な特徴に比例する時間のみで学習できる [Okanohara+ SDM 09]

実験結果 1
精度評価

データ	提案手法	BOW+L_1	SLR	SVM
MOVIE	86.5%	83.0%	81.6%	87.2%
TC300A	80.0%	66.7%	80.0%	73.1%
TC300B	86.7%	86.7%	73.3%	71.9%

- 提案手法が全データセットで高い精度を達成
- BOW+L_1は表現力が非常に低い場合がある
- SLRは近似的にしか最適な部分文字列を抽出できない
- SVMは計算量が訓練データ数の二乗で大きくなり、大規模コーパスに対し適用できない

実験結果2
スケーラビリティ



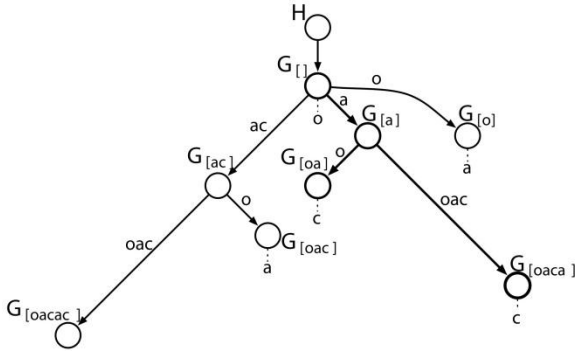
- 全ての部分文字列の統計量計算の時間
- テキストサイズに対し線形

言語モデル

- 文 $S = x_1x_2x_3x_4\ldots$ に対して確率P(S)を与える
  - 音声認識・機械翻訳・スペルチェックなどに使われる
- N-gram モデル(マルコフモデル)
  - N-1文字前までの履歴を使い次の単語を予測
  - $P(x_1x_2x_3x_4\ldots) = \Pi_i P(x_i \mid x_1x_2x_3 \ldots x_{i-1})$
    $= \Pi_i P(x_i \mid x_{i-N+1}x_{i-N+2} \ldots x_{i-1})$
- N-gram モデルの最尤推定
  - $P(c \mid a\ b) = \#(a\ b\ c) / \#(a\ b)$　#()は高発中の出現回数
  - Nが大きくなると推定は非常に粗くなる
- 平滑化
  - 低いオーダーの履歴と線形補間 or バックオフ

Sequence Memoizer [F. Wood+ ICML 09]

- 階層Pitman-Yor過程 [Y. W. Teh ACL 06]の無限履歴長への拡張
- 接尾辞木に基づくGraphical Model
  - 従来の中華料理店過程を利用
  - 接尾辞木中のunary pathで周辺化
- 計算量・作業領域量は従来言語モデルに匹敵
  - 1pathしか更新しないone-shot updateは非常に高速 [J. Gasthaus+ DCC 10]



階層型Pitman-Yor過程 (言語モデルの場合)

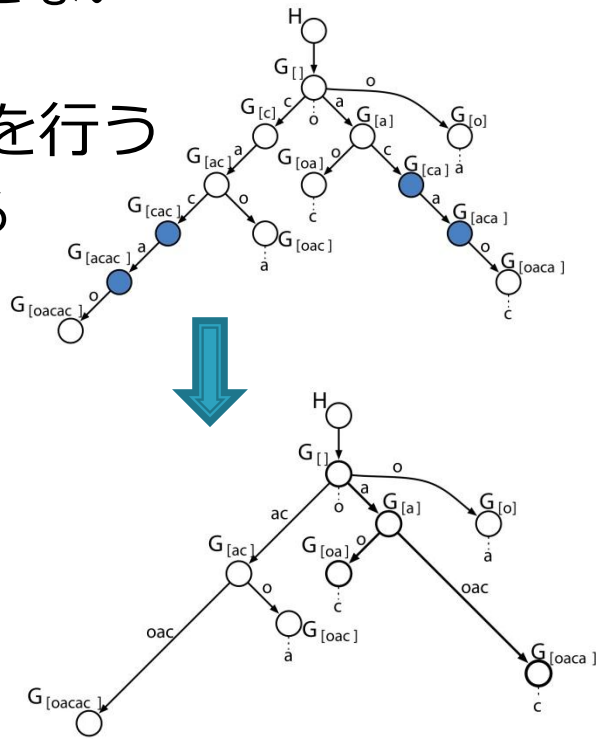
- 短い履歴長の確率分布から、長い履歴長の確率分布を生成する
  - P(z | x y)は P(z | y)から生成されていると思う
  - これらのパラメータは自然に平滑化される
- P(z | x y) ～ PY (d; c; P(z|y))
  - d : Discount
  - c : Concentration
  - P(z | x y)の平均をとると P(z|y)になる
  - d = 0 の時はDirichlet過程
- Kneser-Ney平滑化の一般化
  - KN平滑化は言語モデルで従来の最高精度

Sequence Memoizer

- 予測の際の履歴長を打ち切らない（Non-Markov)

  $P(x_1x_2x_3x_4\dots) = \Pi_i P(x_i \mid x_1x_2x_3 \dots x_{i-1})$

  - 訓練例では $\#(x_1x_2x_3x_4\dots)$ は1回しか出てこないので平滑化は必須
- 階層Pitman-Yor過程を利用して平滑化を行う
  - 適切な履歴長まで勝手に短くしてくれる
  - そのままだと枝数は$O(n^2)$
- Unaryパスをつぶす -> 枝数 $O(n)$
  - 枝を辿るときは次の周辺化を利用
  - もし $G_2|G_1 \sim PY(d_1; 0; G_1)$ であり、$G_3|G_2 \sim PY(d_2; 0; G_2)$ ならば、$G_3|G_1 \sim PY(d_1d_2; 0; G_1)$



図は[Wood+ ICML 09]より

Sequence Memoizer 実験 [Wood+ ICML 09]

AP News コーパス上でのパープレキシティを測定
（低い程良い）

手法	パープレキシティ
4-gram 修正KN	102.4
4-gram 階層Pitman-Yor 過程	101.9
Sequence Memoizer (SM)	96.9

- N-gramの場合、Nを増やすと、性能はSMに漸近するが計算量、ノード数は指数的に爆発する

まとめ

- 大規模文字列を処理するための
  データ構造・アルゴリズムは揃いつつある
  - 高速・スケーラブル・安定
  - 例：100万文書中の全部分文字列の統計量の計算が
    20分弱で 作業領域量は入力サイズと同じ
- 従来考えられなかった問題が解けるように
  - 全ての部分文字列を利用して文書分類
  - 全履歴を利用して次の単語を予測する
- 他の機械学習への応用
  - 木、グラフ、関数の簡潔データ構造

参考文献

- 本発表中に取り上げなかった論文
- 圧縮全文索引
  - V. Makinen and G. Navarro “Compressed Full Text Indexes” ACM Computing Surveys 39(1), article 2, 2007
  - P. Ferragina, R. González, G. Navarro and R. Venturin “Compressed Text Indexes: From Theory to Practice!”

参考文献

- [J. Dean, LADIS 2009] J. Dean, “Large-scale distributed systems at google: Current systems and future directions”,, LADIS 2009
- [Grossi+ SODA 03] R. Grossi, A. Gupta and J.S. Vitter, “High-order entropy-compressed text indexes” SODA 2003
- [U. Manber 91] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches“, *SIAM Journal on Computing*, Volume 22, Issue 5, pp. 935-948.
- [G. Nong+ DCC 09] G. Nong, S. Zhang and W. H. Chan, “Linear Suffix Array Construction by Almost Pure Induced-Sorting” DCC 2009
- [M. Burrows+ 94] M. Burrows and D. Wheeler “A block sorting lossless data compression algorithm”, Technical Report 124, Digital Equipment Corporation 1994
- [Okanohara+ SPIRE 09] D. Okanohara and K. Sadakane “A Linear-Time Burrows-Wheeler Transform using Induced Sorting", SPIRE 2009
- [H. Kaplan+ TCS 07] H. Kaplan, S. Landau, and E. Verbin “A simpler analysis of Burrows-Wheeler based compression” *Theoretical Computer Science* 387:3 (2007), 220-235

- [P. Ferragina+ FOCS 00] P. Ferragina and G. Manzini: «Opportunistic Data Structures with Applications», FOCS 2000
- [Kasai+ CPM 01] T. Kasai, G. Lee, H. Arimura, S. Arikawa and K. Park “Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications”, CPM 2001
- [Okanohara+ SDM 09] D. Okanohara and J. Tsujii "Text Categorization with All Substring Features“ SDM 2009
- [Vishwanathan+ 04] S. V. N Vishwanathan and A. J. Smola, "Fast kernels for string and tree matching", Kernels and Bioinformatics 2004
- [G. Ifrim+ KDD 08] G. Ifrim, G. Bakir and G. Weikum, "Fast Logistic Regression for Text Categorization with Variable-Length N-grams", KDD 2008
- [F. Wood+ ICML 09] F. Wood, C. Archambeau, J. Gasthaus, L. F. James and Y.W. Teh. “A Stochastic Memoizer for Sequence Data.
- “ ICML 2009.
- [Y. W. Teh ACL 06] Y.W. Teh , “A Hierarchical Bayesian Language Model based on Pitman-Yor Processes” Coling/ACL 2006.
- [J. Gasthaus+ DCC 10] “J. Gasthaus and F. Wood and Y.W. Teh “ Lossless Compression based on the Sequence Memoizer” DCC 2010.

予備スライド

高速化の肝：高速なアルゴリズム

- N　文字列長
- σ　文字種類数
- D　文書数
- これらの変数に計算量・作業領域量ともに線形以下で依存するようなアルゴリズムが必須
  - $\Omega(N)$ のアルゴリズムは現実的な時間で動かない
  - ヒューリスティックや枝刈りに頼らない

- 例1：全文書中の全ての部分文字列の出現頻度は $O(N)$ 時間、$N \log \sigma + o(N \log \sigma)$ bitsで計算できる
- 例2：全文書中の全ての部分文字列の文書頻度は $O(N)$ 時間、$N \log \sigma + O(N)$ bits で計算できる

- $N, \sigma, D \sim 10^{10}$ でも1台で動作する

全ての部分文字列の情報は出現情報が同一の$2N-1$個のグループに分けられる

簡潔データ構造
Succinct Data Structure (SDS)

- 作業領域がコンパクトなデータ構造
  - 作業領域はサイズの下限に漸近
  - 圧縮とは違い復元操作は必要ない
  - 操作時間の多くがO(1)
- 様々なデータ構造に対して提案される
  - ビット列/文字列グラフ/連想配列
- 今日紹介するのは次のSDS
  - ビット配列に対するSDS ⇒ Rank/Select辞書
  - 文字列に対するSDS ⇒ Wavelet Tree
  - 木に対するSDS ⇒ LOUDS

ビット配列に対する簡潔データ構造
Rank/Select 辞書

- ビット列 B[1, n]に対し次の操作を備える
  - $rank_c(B, i)$: B[1…i] 中の$c \in \{0,1\}$の数を返す
  - $select_c(B, i)$: i番目の$c \in \{0,1\}$の位置を返す

例： B=0110011100

$rank_1(B,7)=4$

$select_0(B,5)=3$

i	1	2	3	4	5	6	7	8	9	0
	0	1	1	0	0	1	1	1	0	0
	0	1	1	0	0	1	1	1	0	0

Rank/Select辞書の実現 (続)

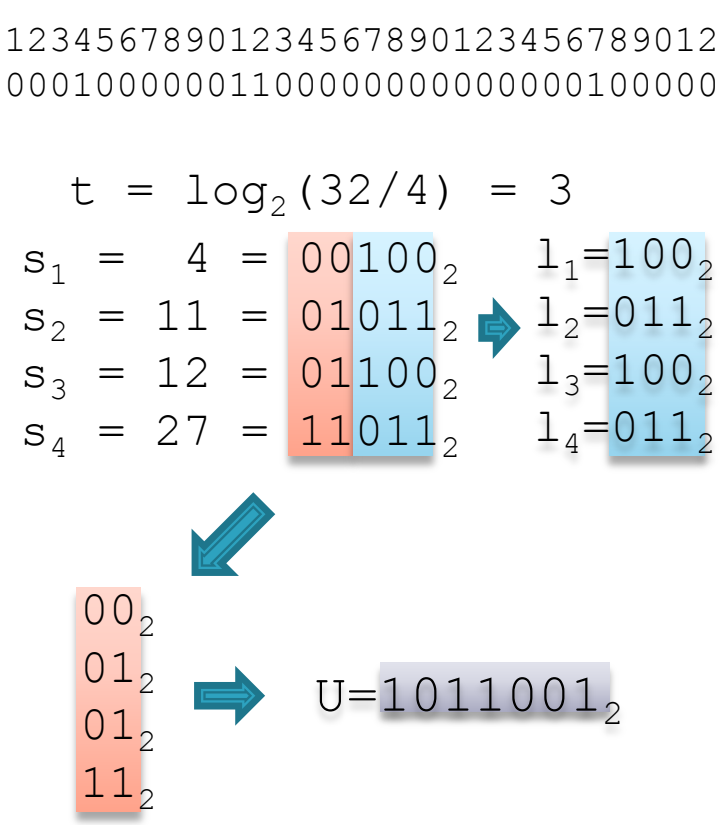
- Rank/Select辞書のサイズの下限 L(n, m)
  - n : Bの長さ
  - m: B中の1の数
- $L(n,m) := log_2\binom{n}{m}$
  - X通りありうるデータ構造を格納するには$log_2 X$ bit必要
- L(n,m) bits O(1)時間のRank/Select辞書を考える
  - $m \cong n/2$の時、 L(n,m) = n + o(n)
    - ビット列 + 補助データ構造
  - $m \ll n/2$の時、 L(n,m) = m(log(n/m) + 1.44) + o(m)
    - ビット列本体は持てない

Rank/Select辞書の実現
$m \cong n/2$ の時

- **rank**の値は大ブロック, 小ブロック, 残りのビット列の合計で求められる
  - これらは全て表引きでO(1)時間で求められる
  - 残りのビット列に対するrankは表EよりO(1)で計算
- **select**の値はブロック単位での二分探索
  - O(log N) 時間
  - O(1)のものもあるが補助データ構造はやや大きい

Rank/Select辞書の実現
$m \ll n$ の時

[Okanohara, Sadakane ALENEX 06]



- $s_i := \mathrm{select}_1(B, i)$, $t := \log_2(n/m)$ とする
- s_i の下位 t bit を l_i とし、そのまま保存
  - サイズ : $m \log_2(n/m)$ bits
- s_i の残りの上位 bitを h_i とおく
  - $0 \leqq h_1 \leqq h_2 \leqq h_3 \ldots h_m \leqq m$ が成り立つ
  - $d_i := h_i - h_{i-1} \geqq 0$ とおく
- ビット列 $U := 0^{d_1}10^{d_2}10^{d_3}\ldots 0^{m}1$ を考える
  - 0^x は0がx個並んでいるという意味
  - U中には1がm個, 0が高々m個ある
- $h_i := \mathrm{select}_1(U, i) - i$ が成り立つ
  - U に対するrank/selectは $m \cong n/2$ のselectを使う

Rank/Select辞書の実現
$m \ll n$ の時

- $\mathrm{select}_1(B, i) = s_i = 2^t h_i + l_i$
  $= 2^t(\mathrm{select}_1(U, i) - i) + l_i$
- $\mathrm{rank}_1(B, i)$
  - $B[1, (i/2^t)*2^t)$中の1の数 = $\mathrm{rank}_0(U, i/2^t) - i/2^t$
  - $B[(i/2^t)*2^t, i]$ 中の1の数 = 対応するrankの数
- サイズ:Uが$2m$, $\{l_i\}_{i=1\ldots m}$が$m\log_2(n/m)$ bit
  $m\log_2(n/m) + 2m + o(m) = L(n,m) + o(m)$

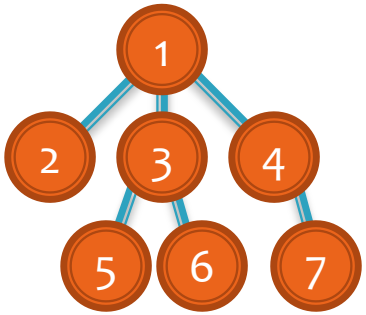
$U=1011001_2$

- 例 $\mathrm{select}_1(B, 3) = 2^3(\mathrm{select}_1(U, 3)-3) + l_3$
  $= 8 + 4 = 12$

$l_1=100_2$
$l_2=011_2$
$l_3=100_2$
$l_4=011_2$

簡潔木

- 木をコンパクトに表現し、そのまま扱いたい
  - ポインタ表現の場合 1ポインタで32～64 bit 必要
- nノードの順序木を2n+o(n) bitで表現
  - nノードの順序木の種類数は約 4^n（C_nカタラン数）
  - 2n bitの表現は下限

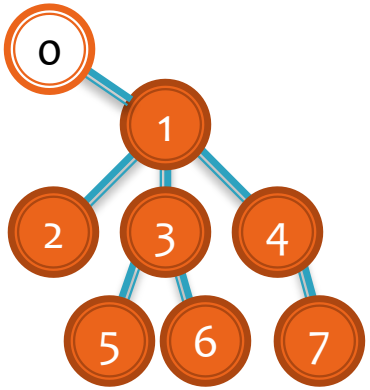


BP表現による例 (LISPなど)
木を深さ優先探索で辿り、入る時に'(', 出る時に')'を記述する

((()(()())(())))

LOUDS: (Level Ordered Unary Degree Sequence)

- 木を幅優先探索(BFS)で辿り、各節点についてk個の子があるならk個の0の後に1個の1を書く
  - 先頭にスーパールート01を書いておく（番人）

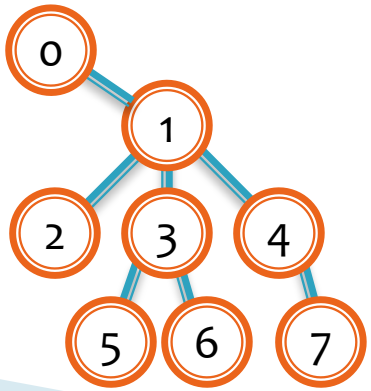


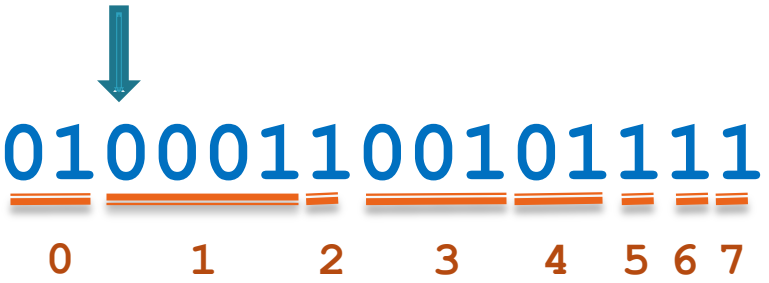
0 1 0 0 0 1 1 0 0 1 0 1 1 1

0 1 2 3 4 5 6 7

LOUDS: (Level Ordered Unary Degree Sequence)

- i番目のノードはi番目の0とi+1番目の1で二回表現される
  - 0は親から指される時、1はその列が終わる時
- ノードは(p, i)で表される
  - p:B中の位置, i:BFS中の番号 (= $rank_1(B, p)$)
- ルート　(3, 1)
- (p, i) の(k+1)番目の子ノード　($select_1(B, i+k)$, p-i+k)
- (p, i) の親ノード　($select_1(B, i+k-1)$, p-i+k)





Graftingによる最適化

[Perkins, JMLR 03]

H = {}　　(有効な特徴集合)

loop

　v := ∂L(**w**)/∂**w**　　(対数尤度の勾配)

　$k^* := \mathrm{argmax}_k |v_k|$　　(勾配が最大の特徴を選択)

　if $|v_{k^*}| < C$ then break (最適解への収束が保証)

　H=H∪{k^*}

　Hのみで$L(\mathbf{w})+C|\mathbf{w}|$を**w**に関して最適化

end loop

output H中の特徴の重み（Hに含まれない重みは全て0）

勾配の絶対値が一番大きい特徴を
効率的に求められさえすれば良い！

それ以外の計算量はO(|H|)

実験
実験データ

データ	訓練例数	単語種類数	全文書長	部分文字列種類数	MS数
MOVIE	2000	38187	7.8×10^6	1.5×10^{11}	1.7×10^6
TC300A	200	16655	2.0×10^6	1.9×10^{11}	3.7×10^5
TC300B	200	14430	2.4×10^6	1.5×10^{11}	2.3×10^5

- 文書の二値分類タスク
  - Movie：　極性評判分析用の実験データ
  - TC300A/B：TechTC-300の文書分類タスクセット中で、SVMの精度が7割の二つ
    (分類に有効な部分文字列は1, 2個)

実験
評価手法

- 10分割交差検定の正解精度
- 提案手法
  - 真面目なGraftingではなく、毎回1000個特徴候補を取り出し、候補に加える．素性値にはidfを利用した
- BOW+L_1
  - スペースで区切られた部分文字列を単語として扱いL_1+LRで学習（OWL-QN)
  - 表記ゆれなどは吸収
- SLR [G. Ifrim, et. al. KDD 08]
  - 部分文字列をgreedyに探索してLRで学習
- BOW+SVM
  - 多項式カーネル（文字列カーネルは動かなかった）

L_1正則化付最尤推定による学習

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_i L(\mathbf{w}) + C\,|\mathbf{w}|_1$$

$$L(\mathbf{w}) = -\sum_i \log p(y_i \mid x_i; \mathbf{w})$$

L_1正則化

- 訓練データ $\{(x_i, y_i)\}$ $(i=1 \ldots N)$を用いて**w**を推定
  - $|\mathbf{w}|_1 = \Sigma_i |w_i|$，$C>0$はハイパーパラメータ.
- L_1正則化による疎なパラメータが得られる
  - 特徴選択の効果があり、多くのw_iが0になる
  - 有効な重み数はL_2正則化の場合よりはるかに少ないが精度はほぼ変わらない [J. Gao, ACL 06]