2005年12月作成

圧縮索引とその周辺

岡野原　大輔

hillbig@is.s.u-tokyo.ac.jp

東京大学

発表の流れ

- **背景**
  - **索引 / 全文索引 / 圧縮索引**
- **基礎知識** 1990年代
  - Suffix Arrays, Burrows Wheeler Transform
- **圧縮全文索引** 2000年
  - Compressed SA, FM-index
- **最近の話題** 2005年～
  - Wavelet Tree, XWT (Tree**の**BWT)
  - Succinct**なデータ構造** (bit array, tree)

背景

大規模データの利用

- **非常に大きなテキストデータが手に入るようになった**
  - MEDLINE (1100**万アブストラクト** 500GB)
  - Blog Watcher (1100 blog **エントリー**)
  - TREC2004 Terabyte Track (2500**万文書** 426GB)
  - Web Pages in Internet (**〜** PB)
  - Genome **配列** (> 800G **塩基対** in 2004)
- 「*We can obtain accurate information from very large inaccurate data (e.g. 50% is error) by central limit theorem*」 (Mehran Shaami@AAMT2005)
  - c.f. Google**の大規模なウェブデータを使った統計的機械翻訳システムは**2005**年のワークショップで最高精度。**

問題設定

- 前もって与えられたデータ T (長さ:n アルファベット集合: Σ) とパタンP (長さm)
- 次の二つの操作が答えられるか
  - occ(P) パタンPのデータ中の出現回数
  - loc(P) パタンPのデータ中の全ての出現位置
  - 他の多くの操作もこれらの基本操作の組み合わせに帰着される

用語

- 索引
  - パタンの出現した回数、場所を前もって求め保存したもの
- 全文索引
  - 全てのテキスト中に出現したパタンの出現した回数、場所を前もって求め保存したもの
  - 文字索引 (Suffix Arrays, Suffix Trees, N-gram など)
    - 任意の文字列を検索可
  - 単語索引 (転置ファイルなど)
    - 形態素解析等で抽出された単語の出現位置のみ記録
- 圧縮全文索引
  - 全文索引を圧縮したもの。復元操作は、前もって行われない
- 本発表では、圧縮全文索引の中で文字索引を扱う

なぜ索引？

- **大規模データを利用するには線形時間操作でさえコストが大きい**
  - **例　データ**1G**中の**“the”**の出現場所を全て報告**
    - O(logN) (suffix arrays) 0.005 msec
    - O(N) (grep)　　970　msec
- **索引を使うことで、パタン**P**の出現位置や回数を高速に求めることが可能**

なぜ文字索引？

- 文字索引：任意のパタンPに対し答えられる
- 従来の転置ファイルは答えが漏れる場合がある
- 日本語等、分かち書きでない言語では特に問題。英語でもフレーズを探索できない
(統計的機械翻訳とかで問題)
- ゲノムなど単語が無い場合はさらに困難
- N-gram方式（N文字転置ファイル）が最近利用されている
  - この場合、辞書ヒット数が大きい場合、計算量はO(N)に近づくため、大規模データでは使えない

なぜ圧縮全文索引？

- 索引では速度と作業領域量はトレードオフ関係
  - 極端な話、全ての答えを前もって求めおくと
    計算量はO(1)だが、作業領域量は非常に大きい
- 全文文字索引では、作業領域量が非常に大きい
  - 扱えるデータサイズに制限があった。
- このトレードオフはどっちを重視すればいいの？

  →作業領域量、速度の両面で*ほぼ*最適なものが達成可能
  - 最新の結果: NH_kbitの作業領域量でocc(P)をO(m) time、[Ferragina 2005]
    - H_k : 入力Tのk次エントロピー。
  - 実データのH_5の例　英文:0.23　DNA:0.24　XML:0.10

圧縮索引の応用例

- **情報抽出**
  - **固有表現抽出** $df_2(P)/df(P)$ [Church 2001]
- **生物情報**
  - **ゲノム解析** (**ゲノム比較**、**ターゲット予測**)
- **機械学習**
  - N-gram feature**の利用** (boosting**など**)
  - **木構造の索引** (tree kernel)
- **統計的機械翻訳**
  - **フレーズアライメント**
  - **言語モデル**

基礎知識

Suffix Arrays
Burrows Wheeler Transform

Suffix Arrays (SA) [Manber 1989]

- **入力**: $T=t_1 t_2\, t_3..t_N$
- **Tの接尾辞**(suffix): $S_k= t_k\, t_{k+1} t_{k+2}..t_N$

S_1	abraca$	S_7	$	7
S_2	braca$	S_6	a$	6
S_3	raca$	S_1	abraca$	1
S_4	aca$	S_4	aca$	4
S_5	ca$	S_2	braca$	2
S_6	a$	S_5	ca$	5
S_7	$	S_3	raca$	3

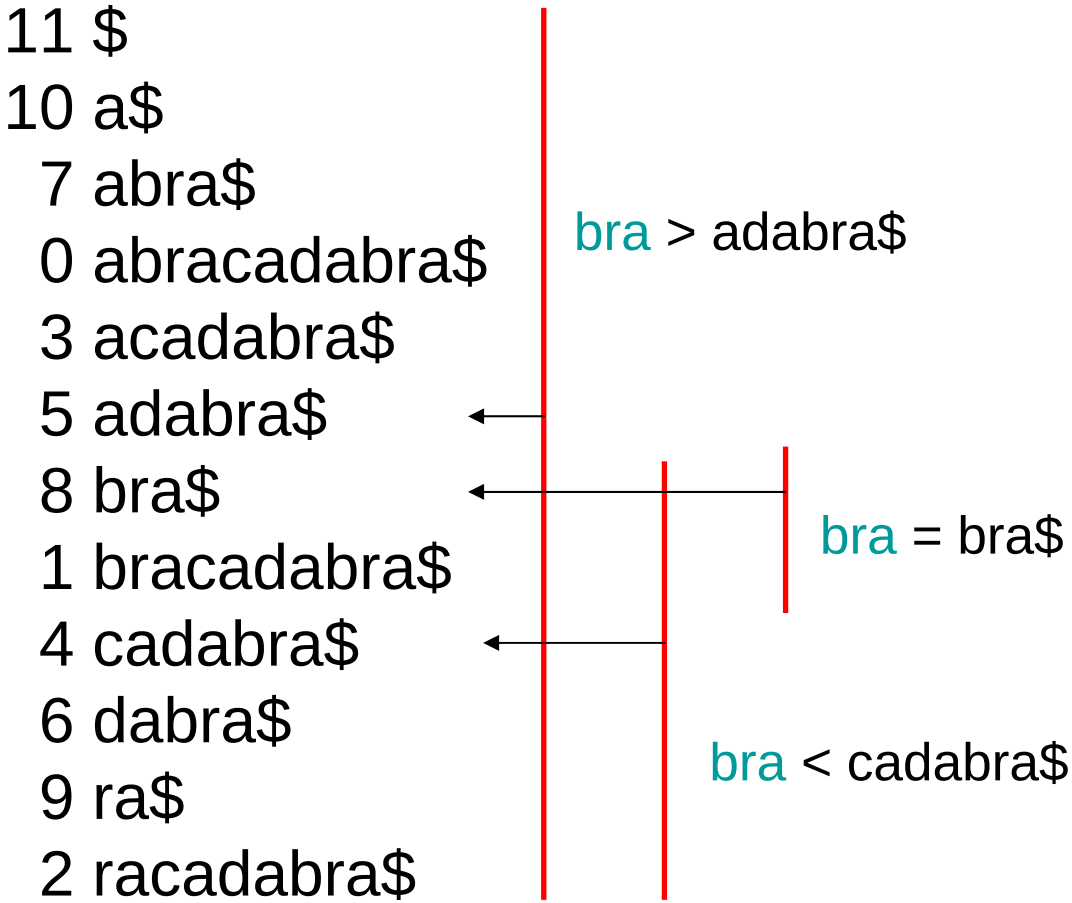
(1) **Tの全ての接尾辞を列挙**

(2) **接尾辞集合を辞書式順序でソートする**

(3) **接尾辞の番号を抽出**

SAを使った検索

入力 T=abracadabra$ **パタン** P = bra



時間計算量

occ(P):
O(m log n)

loc(P):
O(m log n + occ(P))

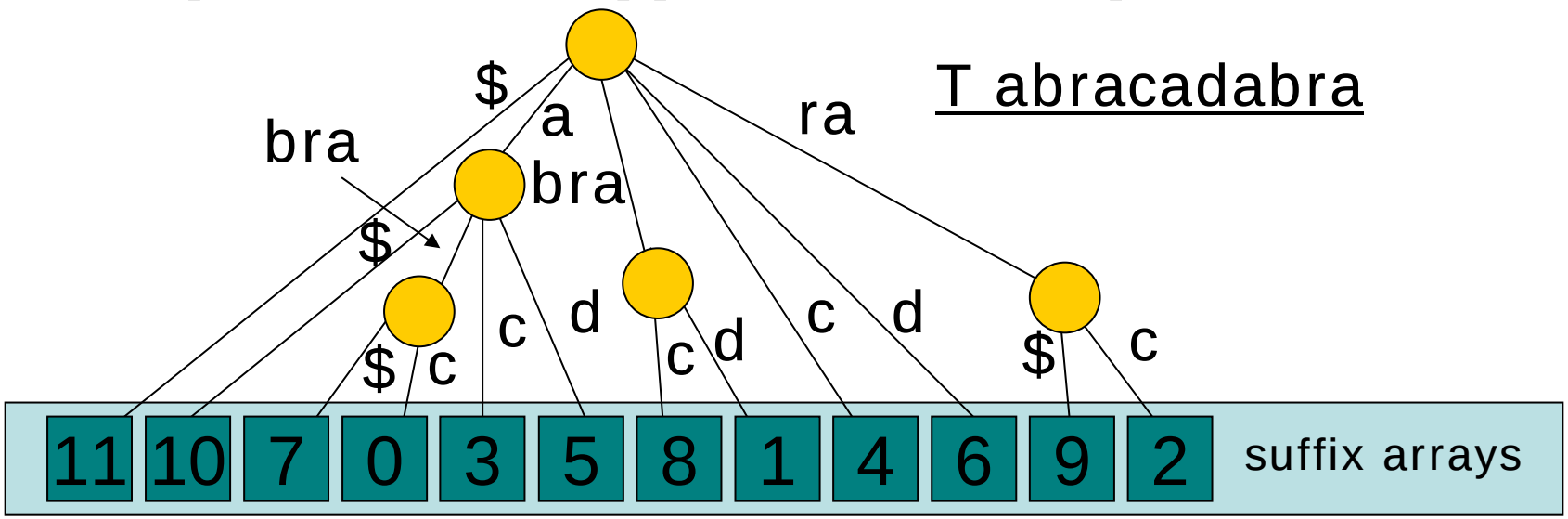
空間計算量

log n bit (4n byte)

Hgt**配列を使うと**
occ(P)**は**O(m+log n)

C.f. Suffix Trees (ST) [Weiner 1973]

- **Tの全Suffixからなる**Trie**の圧縮 (縮退) 表現**
- **多くの文字列アルゴリズムが**ST**上で動く**
- **非常に作業領域量が大きい**
  - Compressed ST**は約**9n bit
    [Sadakane 2004] [Okanohara 2005]



Burrows Wheeler's Transform [1994] (BWT)

- **文字列に対する可逆変換**
  - **最初は**bzip2**等の可逆圧縮に使われていた**
- BWT[i] := T[SA[i]-1]
- abracadabra$ ⇒ $_{BWT}$ ard$rcaaaabb
- BWT**を適用したテキストは非常に圧縮しやすい**
  - **同じ文脈の直前には同じ文字が現れやすい**

  t hese are great ...

  t hese are possible ...

  t hese were not of ..

  t hese ...

例 BWT前

When Farmer Oak smiled, the corners of his mouth spread till they were within an unimportant distance of his ears, his eyes were reduced to chinks, and diver gingwrinkles appeared round them, extending upon his countenance like the rays in a rudimentary sketch of the rising sun. His Christian name was Gabriel, and on working days he was a young man of sound judgment, easy motions, proper dress, and general good character. On Sundays he was a man of misty views, rather given to postponing, and hampered by his best clothes andumbrella : upon the whole, one who felt himself to occupy morally that vast middle space of Laodiceanneutrality which lay between the Communion peopleof the parish and the drunken section, -- that is, he wentto church, but yawned privately by the time the con+gegation reached the Nicene creed,- and thought ofwhat there would be for dinner when he meant to belistening to the sermon. Or, to state his character asit stood in the scale of public opinion, when his friendsand critics were in tantrums, he was considered rather abad man ; when they were pleased, he was rather a goodman ; when they were neither, he was a man whose moral colour was a kind of pepper-and-salt mixture.Since he lived six times as many working-days as Sundays, Oak's appearance in his old clothes was mostpeculiarly his own -- the mental picture formed by hisneighbours in imagining him being always dressed inthat way. He wore a low-crowned felt hat, spread outat the base by tight jamming upon the head for securityin high winds, and a coat like Dr. Johnson's ; his lowerextremities being …..

例 BWT後

oooooioororooorooooooooooorooorromrrooomoorooooooooormoorooororiooorooor
mmmmmuuiiiiiIiuuuuuuuuiiiUiiiiiiooooooooooooorooooiiiiioooioiiiiiiiiiiiioiiiiiieuiiiiiiiiiiiiiiii
iouuuuouuUUuuuuuuoooouuiooriiiriirriiiiriiiiiiaiiiiioooooooooooooooiiiouioiiiioiiuiiuiiiiiii
iiiiiiiiiiiiiiioiiiiioiuiiiiiiiiiiiiioiiiiiiiiiiiiioiiiiiuiiiioiiiiiiiiiiiiioiiiiiiiiiioiiiioiiiiiiioiiiaiiiiiiiiiiiiiiiiiiioiiiiiioiiiiii
iiiiiiiiuiiiiiiiiiiiiiiiiiiiioiiiiiiioiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiuuuiioiiiiiuiiiiiiiiiiiiiiiiiiiiiiiiioiiiiuioiuiiiiiiioiiiiiiui
iiiiiiiiiiiiiiiiiiiiiiiiiiiiioaoiiiiioioiiiiiiiioooiiiiiooioiiioiiiiiouiiiiiiiiiiiiooiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioiiiiiiiiiiiiiii
iiiioiooiiiiiiiiiiioiiiiiuiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioiiiiiiiiiiiiioiiiuiiiiiiiiiiioiiiiiiiiiiiiuoiiioiiioiiiiiiiiiiiiiiiiiiiiiiiu
iiiiuuiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiuiuiiiiiuuiiiiiiiiiiiiiiiiiiiiiiuiiiiiiiiiiiiiiiiiiiiiiiiiiiioiiiiiiioiiiiiiiiiiiiiiiiiiiiiiioiiiiiii
ioiiiiuiiiioiiiioiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioiioiiiiiiuiiiiiiiiiiiiiiiiooiiiiiiiiiiiiiiiiiiiiioooiiiiiiiioiiiiouiiiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioiiioiiiiiiioiiiiiioiiiiuiuoiiiiiiiiioiiiiiiioiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioioiiiiiiooiiiiiiiiiiiiii
oiiiiiiioiiiuiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioiiiiiuiiiiiiiiiiuoiiiiooiiiiiiiiiiioioiiiioi
ooiiioiiiiiiiiiiiiiuiiiiiiiuiiiiiiiiiiiiiiiiiiiiioiiuiiiiiiiiiiiiiiiiiiiiiiiiuiiiiiuiiiiiiiiiioiiiiiiiiiiiioiiiiiiiiiiiiuuioiiiiiiiiiiioii
iiiiiiiiiiiiiouiiiioioiiiiiiiiiiiiiiiiioioiiiiiuuuuiiiiiiiiuiiuiiiiiiiuiiiiuiiiiiiiiiiiiiiiioiiiiiiiiiiiiiiiioiioiiiiiiiiiiiiiiii
iiiiiioiiiiiiiiiiiiiiiiiiuuuiiiiiiiuioiuuuiiiiiiiiiiiiuiiiiiiiiiiiiiiiiiioiiiiiiiiiiiiiiioiiiiioiiiiioiiiiiiiiiiiiiiiiiiiiiiaiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiiiiioiiiiiouoiiiiuuuiiioiiiiiiioiiiiiiiiiiioiiiiiiiiiiiioiiiiiiiiiiioiiioiooiiiiiiiiiiioiiiiiiiiiiiiiiiiiiiiiiiiiui
iiiiiiiiiiiiiiiiiiiiiiiiuiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioiiioiiiiiiiiiiiiiiiiiiiiiiiiiuiiiiiiiiiiiiiiiiiiiiiiuiiiiiiiiiiiiiiiii
iiiioiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiuiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiuioiiii
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioiiiiiiiuiiiiiiiioiiiioiiiiiiiiiiiiiiiuiiiiiiiiiiiiiiuiuiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiaiiiiiiiiii
iiiiiiiioiiiiiiiiiiioiiiiioiiiiiiiaaiiiiiiiiiiiiiiiiuiiiiiiiiiiiiiiiiiiiiiiiiiiioiiiiiiiiiiiiiiiiioioiiiiiiiiiiiiiiiiiiiiiiiiuuuuiuiiii
iiiiiiiiiiiiiiiiiiiiiiioiiiiiiiioiiiiiiiiiiiiiiiiiiiuioiiuuuuuiuuiiuuiuuiiuuuuuuuiuuuiuuuuiiiiuuuiiiiiiiiiii
iiioiiiiiiiiiiiiiiiiiiuuoiiiiaiouiiiuiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioiiiiiiiiiiiiiiiiiiioiiiioiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
…..

BWTの重要な性質

- **同じ文字のBWT中に出現する順番はFでも同じ順番**
- BWT，Fの順序はそれに続く文字列の順位で決まる。どちらも同じ文字列が順位決定に使われる
- BW逆変換および、検索でこの性質を利用

a_1	$
r	a_1 $
d	a_2 bra$
$	a_3 bracadabra$
r	a_4 cadabra$
c	a_5 dabra$
a_2	bra$
a_3	bracadabra$
a_4	cadabra$
a_5	dabra$
b	ra$
b	racadabra$

BWT

F

index	SA	BWT	F	Suffix
0	11	a_1	$	$
1	10	r_1	a_1	a$
2	7	d	a_2	abra$
3	0	$	a_3	abracadabra$
4	3	r_2	a_4	acadabra$
5	5	c	a_5	adabra$
6	8	a_2	b_1	bra$
7	1	a_3	b_2	bracadabra$
8	4	a_4	c	cadabra$
9	6	a_5	d	dabra$
10	9	b_1	r_1	ra$
11	2	b_2	r_2	racadabra$

BWT後のテキストから
元のテキストを復元する。

BWTの性質（同じ文字の順番は
BWT中でもFでも変わらない）を
利用し、データをたどっていく。

F[i]に対応するBWT中の
位置をLFmapping[i]とする。
例 LFmapping[3]=7
　　LFmapping[7]=11

$の位置を求める　i’ = 3
a_3 を出力
a_3 のBWT中の位置を求め(7)
それをi’に代入 i’ = 7
b_2を出力
b_2 のBWT中の位置を求め(11)
それをi’に代入 i’ = 11
。。

逆BW変換 (LF-mapping)

```
void revBWT(char* bwt, int n){
     int count [0x100];
     memset(count,0,sizeof(int)*0x100);
     for (i = 0; i < n; i++) count[bwt[i]]++;
     for (int i = 1; i < 0x100; i++) count[i]+=count[i-1];
     int* LFmapping = new int[n];
     for (int i = n-1; i >= 0; i--){
           LFmapping[--count[bwt[i]] = i;
     }
     int next = find(BWT,'$'); //find the position of '$'
     for (int i = 0; i < n; i++){
           next = LFmapping[next];
           putchar(bwt[next]);
    }
     delete[] LFmapping;
}
```

最近のSAの話題

- Compressed Suffix Arrays (CSA) **後で詳しく**
- **高速な構築**
  - **線形時間構築** [Ko 03] [Kim 03] [KS 03]
  - **現実的に一番速いものはこれらでは無い**
    c.f. dssort, msufsort, divsort
- **メモリ効率の良い構築方法**
  - CSA**を直接構築** [sadakane 03]
  - CSA**の高速構築** [mori 05] [okanohara 05]
    - **後で述べる**wavelet tree**を利用**
- **近似マッチング** [Huynh 05]

圧縮全文索引

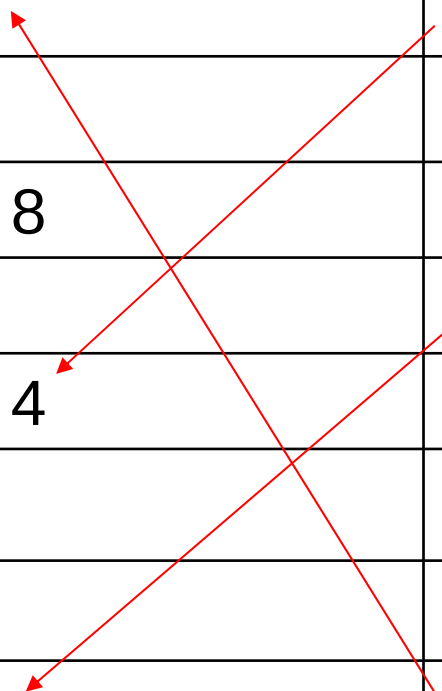
Compressed Suffix Arrays (CSA)

[Grossi, Vitter 00][Sadakane 03][Grossi, Guputa, Vitter 03]

- **元のSAの作業領域量は** nlogn bit
  - **SAは1からNまでの並び替えなので冗長性が無く圧縮はそのままでは不可能**
- **SAの代わりに次のように定義されるΨを保存**
  - **Ψ**[i] = SA^{-1}[SA[i] + 1]
  - SA^{-1}[i] = j　s.t.　SA[j] = i
  - **SAをサンプリングした**SA_k[i]=SA[ik]**も一緒に保存**
- **Ψを使ってSA上を移動。**SA_k**からSAを求める**
  - SA[i] = SA[**Ψ**[i]]-1 = SA[$\mathbf{\Psi}^2$[i]]-2 = ... = SA[$\mathbf{\Psi}^n$[i]]-n
- If SA[$\mathbf{\Psi}^n$[i]] = p　then SA[i] = p-n

I	SA	SA^{-1}	$\Psi := SA^{-1}[SA[i]+1]$	Suffix
0	11	3	3	$
1	10	7	0	a$
2	7	11	6	abra$
3	0	4	7	abracadabra$
4	3	8	8	acadabra$
5	5	5	9	adabra$
6	8	9	10	bra$
7	1	2	11	bracadabra$
8	4	6	5	cadabra$
9	6	10	2	dabra$
10	9	1	1	ra$
11	2	0	4	racadabra$

I	sampled SA	$\Psi := SA^{-1}[SA[i]+1]$	Suffix
0		3	$
1		0	a$
2		6	abra$
3	0	7	abracadabra$
4		8	acadabra$
5		9	
6	8	10	
7		11	
8	4	5	
9		2	
10		1	
11		4	racadabra$



SA[7]
=SA[Ψ[7]]-1 = SA[11]-1
=SA[Ψ[11]]-2 = SA[4]-2
=SA[Ψ[4]]-3 = SA[8]-3
=4 - 3
=1

$\Psi[i] = SA^{-1}[SA[i] + 1]$ は一体何者？

- **直感的意味**
  - **i番目のsuffixより1文字短いsuffixの順序**
  - **SA表現では辞書隣接情報があり、位置隣接情報がない**
    位置表現では位置隣接情報があり、辞書隣接情報がない
  - **Ψはそれら二つを結び付ける**
- **定理**:
  もし $T[SA[i]] = T[SA[i+1]]$ **ならば** $\Psi[i] < \Psi[i+1]$
- **証明**:
  $T[SA[i]] = T[SA[i+1]]$**ならば**$SA[i]$**と**$SA[i+1]$**の順位はそれらより一文字短い**Suffix**の順位で決定される (辞書式順序)**

  $S_{SA[i]+1} < S_{SA[i]+1} \longrightarrow SA^{-1}[SA[i]+1] < SA^{-1}[SA[i+1]+1]$

  $\longrightarrow \Psi[i] < \Psi[i+1]$

I	SA	SA^{-1}	Ψ := SA^{-1}[SA[i]+1]	Suffix
0	11	3	3	$
1	10	7	0	a$
2	7	11	6	abra$
3	0	4	7	abracadabra$
4	3	8	8	acadabra$
5	5	5	9	adabra$
6	8	9	10	bra$
7	1	2	11	bracadabra$
8	4	6	5	cadabra$
9	6	10	2	dabra$
10	9	1	1	ra$
11	2	0	4	racadabra$

単調増加列

Ψの圧縮

- **Ψは部分単調増加列なので圧縮可能**
  - **差分列dを次のように定義** d[i] = **Ψ**[i+1]-**Ψ**[i]
- delta**符号を利用して**d[]**を圧縮。サイズは** $O(NH_0)$[Sadakane 2003].
- Wavelet Tree**を利用して**d**を圧縮。サイズは** $O(NH_k)$ [Grossi 2003]
- **実用的には、** $\Psi_k[i] = \Psi[ik]$ **とd を保存**
  Ψ[i]**を求めるには、**t = i/k　r = i%k　**であるとき**
  Ψ[i] = Ψ_k[tk]+d[tk]+d[tk+1]+…+d[tk+r-1]

Self indexing property of CSA [Sadakane 2003]

- CSA **は検索時に元テキストを必要としない！**
  - T[i…j] **を復元する** substr(i,j) **もサポート**
- Suffix**の先頭文字は容易にわかる**
  - **各文字の出現回数を保存しておけばよい**
  - D[i] := **テキスト中に実際に出現した文字**
  - C[i] := D[i]**より小さい文字のテキスト中の出現回数**
- T[SA[i]] = D[j] s.t. C[j] ≦ i < C[j+1]

```
void substr(int i, int j) {
    for (int p = SA^-1[i]; i < j; i++, p=Ψ[p]){
        int t = binarySearch(C);
        output(D[t]);
    }
}
```

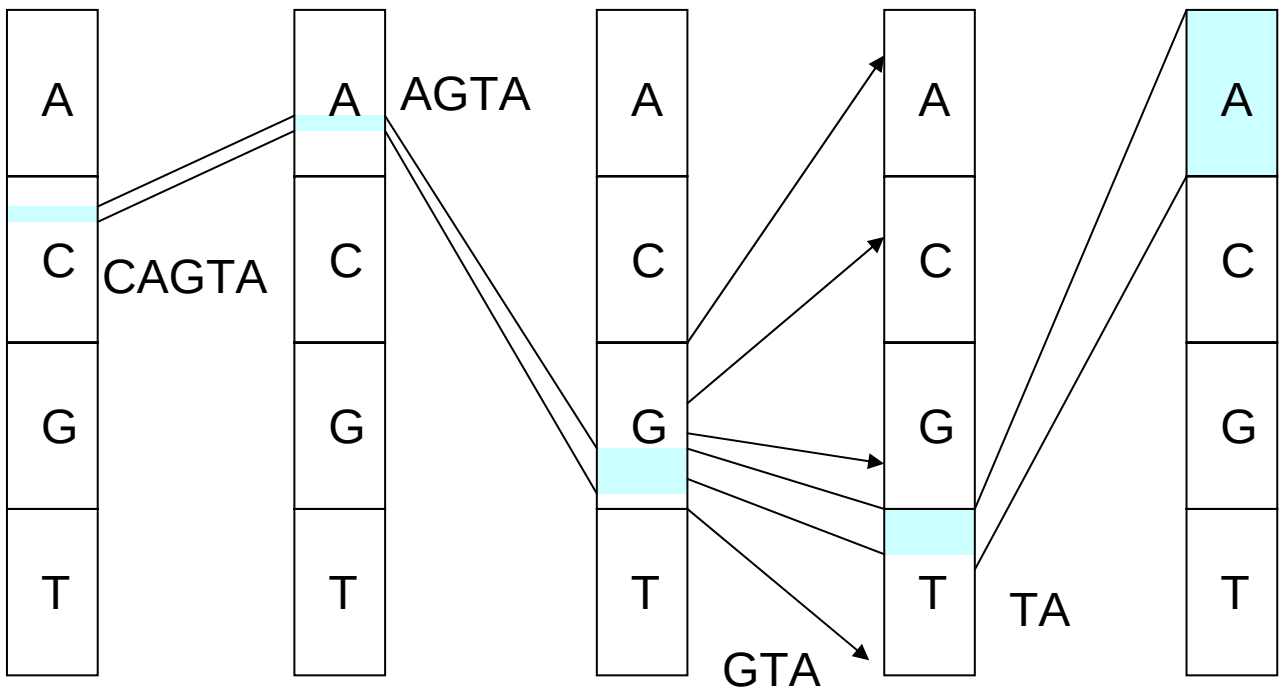
I	SA	SA^{-1}	**Ψ**	Head of Suffix
0	11	3	3	$
1	10	7	0	a
2	7	11	6	a
3	0	4	7	a
4	3	8	8	a
5	5	5	9	a
6	8	9	10	b
7	1	2	11	b
8	4	6	5	c
9	6	10	2	d
10	9	1	1	r
11	2	0	4	r

C[0] = 0
C[1] = 1
C[2] = 6
….
D[0] = ‘$’
D[1] = ‘a’
D[2] = ‘b’

•substr(3,6)
p := SA^{-1}[3] = 4
decode[p]; // output ‘a’
p = **Ψ**[p] = 8
decode[p]; // output ‘c’
p = **Ψ**[p] = 5
decode[p]; // output ‘a’
p = **Ψ**[p] = 9
decode[p]; // output ‘d’

Backward Search [Sadakane 2002] [Makinen 2004]

- **文字列探索時にSAを使うが、SAのlookupは計算量が大きい。Ψだけを用いて探索が可能**
- Search P=CAGTA in backword (P[m] P[m-1]…)



FM-index [Ferragina 2000]

- もう一つの圧縮索引
  - 提案時は、実装が難しいとされていたが、最近現実的な実装方法が次々と見つかってきた
  - ちなみにLZ-indexという圧縮索引もある
- CSAと同様にocc, loc, substrをサポート
- BWTを基にしている
  - LF-mappingを検索に用いる

基本操作の定義

- rank (B, p, x)
  - B[0…p]中のxの出現回数を返す
- select (B, r, x)
  - B中でr回目のxの出現位置を返す
- 例
  - B=d#rcaaaabb
    rank(B,6,a)=2
    select(B,4,a)=7
    select(B,2,b)=9
- rank, select操作を使ってoccを行う

occ(P[1…m],BWT[1…n])

C[*c*]:= c**より小さい文字の出現回数**
BWT T**の**BWT**後のテキスト**

```
1.  i := m
2.  sp := 1; ep := n;
3.  while (sp ≦ ep) and (i >= 1) do
4.       c := P[i];
5.       sp := C[c] + rank(BWT,c,sp-1)+1;
6.       ep := C[c] + rank(BWT,c,ep);
7.       i--;
8.  if (ep < sp)  return 0;
    else          return ep-sp+1;
```

I	SA	BWT	Head of Suffix
0	11	a_1	$
1	10	r_1	a_1
2	7	d	a_2
3	0	$	a_3
4	3	r_2	a_4
5	5	c	a_5
6	8	a_2	b_1
7	1	a_3	b_2
8	4	a_4	c
9	6	a_5	d
10	9	b_1	r_1
11	2	b_2	r_2

occ(“ab”,”ard$rcaaaabb”)

(1) “b”**の領域を求める**

I	SA	BWT	Head of Suffix
0	11	a_1	$
1	10	r_1	a_1
2	7	d	a_2
3	0	$	a_3
4	3	r_2	a_4
5	5	c	a_5
6	8	a_2 ←	b_1
7	1	a_3 ←	b_2
8	4	a_4	c
9	6	a_5	d
10	9	b_1	r_1
11	2	b_2	r_2

occ(“ab”,”ard$rcaaaabb”)

(1) ‘b’**の領域を求める**

(2) **現在の領域の直前である**
(a_2 a_3) **の領域を求める**
(rank(BWT,‘a’,ep),
rank(BWT,‘a’.sp))**を求める**

I	SA	BWT	Head of Suffix
0	11	a_1	$
1	10	r_1	a_1
2	7	d	a_2
3	0	$	a_3
4	3	r_2	a_4
5	5	c	a_5
6	8	a_2	b_1
7	1	a_3	b_2
8	4	a_4	c
9	6	a_5	d
10	9	b_1	r_1
11	2	b_2	r_2

FMcount(“bra”,”ard$rcaaaabb”)

(1) ‘b’**の領域を求める**

(2) **現在の領域の直前である** (a_2 a_3) **の領域を求める**
(rank(BWT,‘a’,ep),
rank(BWT,‘a’.sp))

(3) “ab”**の出現回数は**
(3 - 2 + 1) = 2 **回**

CSAとFM-indexの関係

- CSAはΨを、FM-indexはΨ^{-1}を利用
  - Ψ^{-1}[i] = SA^{-1}[SA[i]-1] = C[T[i]]+ rank(BWT,T[i],i);
  - 別に逆を使っても問題ない
- 本質的に違うのはFM-indexがΨ^{-1}を明示的に持たずにBWTのrankで実現しているところ
  - rank(BWT,c,i)の効率良い実装は存在するか？
  - cが2値の場合は簡単。そうでない場合は困難

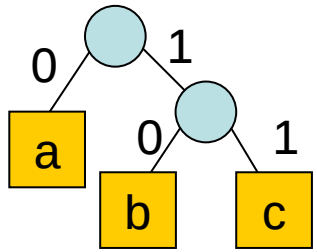
  → 一般のcに対する実装方法が近年提案される

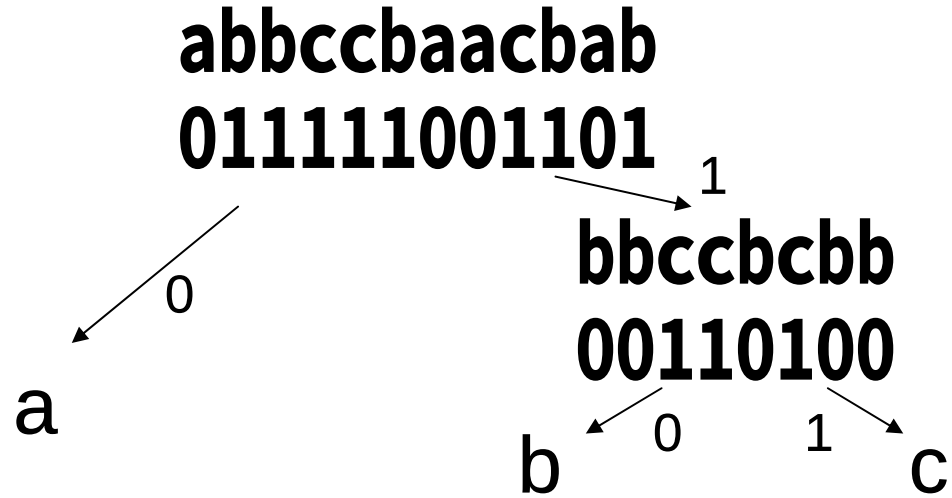
Rank, Select

- **Bが2種類のアルファベットからなる場合**
  - rank(B,i,x), select(B,i,x) **を約**1.5Nbit**の作業領域で**O(1)**時間で実現可能**
  - Rank**はテーブル参照** + popCount
  - Select**は**rank**の問題に帰着させる**
- **Bが3種類以上のアルファベットからなる場合**
  - Wavelet Tree**が現実的** [Grossi 2003]
  - rank, select **を** $O(NH_0)$ **作業領域、**$O(H_0)$**時間で実現**
  - **理論的には**$O(NH_0)$ **作業領域**, O(1)**時間で実現可能** [Ferragina 2005]

Example of Wavelet Tree

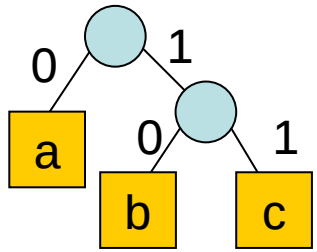
- **Σ** ={a,b,c}
  $a = 0_2$ $b = 10_2$ $c = 11_2$
- T= abbccbaacbab



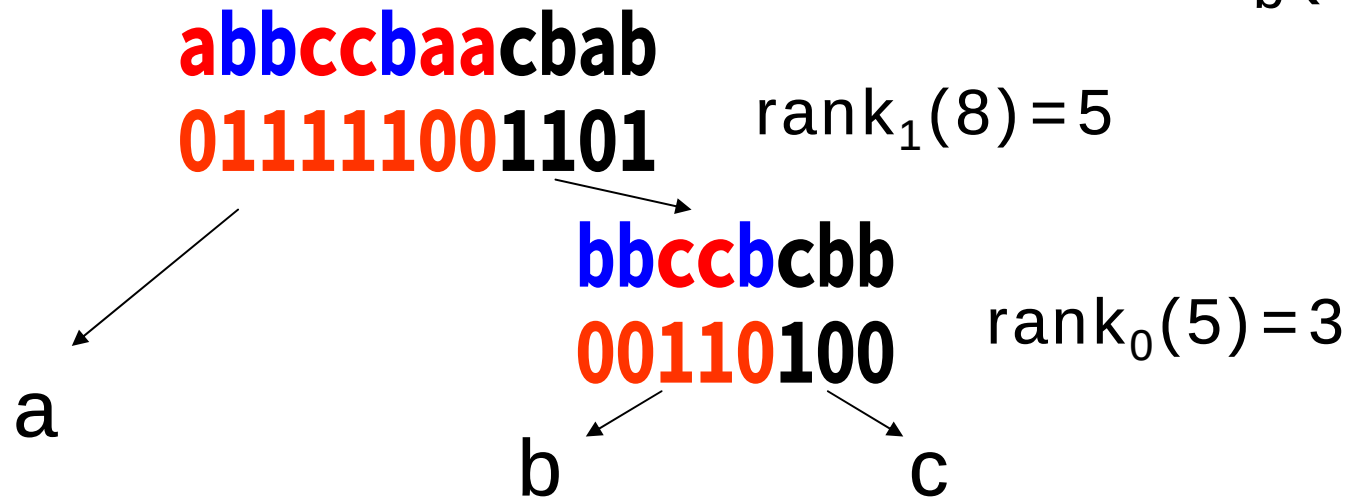


Example of Wavelet Tree (contd.)

- **Σ** ={a,b,c}
  $a = 0_2$ $b = 10_2$ $c = 11_2$
- T= abbccbaacbab



$rank_b(8) = 3$



最近の話題

Succinct*なデータ構造
Bit Array / Tree

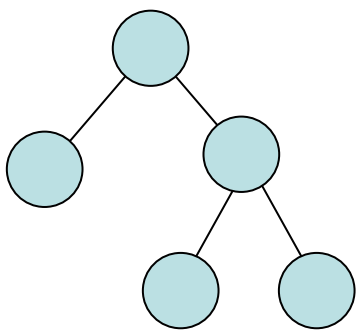
* Succinct : n個の要素をO(n)（もしくは情報理論的な限界に漸近的に近い）作業領域で各操作を定数時間で実行

Succinct Bit Array

- Bit Vector B[1…N]
  - rank (B, p, x) (x=1,0)
  - select (B, r, x) (x = 1,0)
  - **圧縮索引の様々な場面で利用**
    - **サンプリングした位置が**1**、それ以外**0**の**bit vector
- **定数時間**rank/select**で作業領域は**1.5Nbit**程度** [Kim 2005]
- **定数時間**select,O(logN)**時間**rank,**作業領域**O(H_0) [Okanohara 2005]
  - **定数時間**select/rank**で作業領域**O(H_0)**は現在作業中**

Balanced Parenthesis Tree (PT)
[Jacobson 85] [Munro 01] [Geary 05]

- **節点数+葉数=nの木構造を**2nbit**の作業領域で表現**
  - parent, first-child, sibling**の操作時間は**O(1)
  - c.f **情報理論的限界は**2n - o(n)bit
    一般的に木はポインタを利用しnlogn bit**で表現**
  - LISP**と同様に木を**DFS**で辿った時、最初に辿った時’(‘、出る時’)’を出力した括弧列からなる**

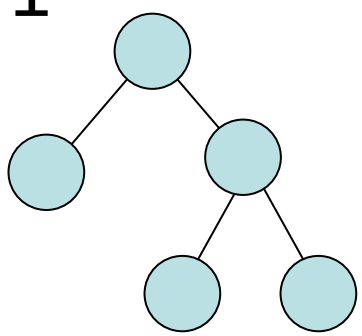


((()(()())))

0010010111

Balanced Parenthesis Tree (続)

- **括弧列上の次の操作を定数時間で行う**
  - findopen(*x*), findclose(*x*): x**に対応する括弧の位置を返す**
  - enclose(*x*): x**を最もきつく囲む括弧対の開き括弧の位置を返す**
- **木の操作は次のように実現**
  - parent(x) = enclose(x)
  - sibling(x) = findclose(x)+1
  - first-child(x) = x+1

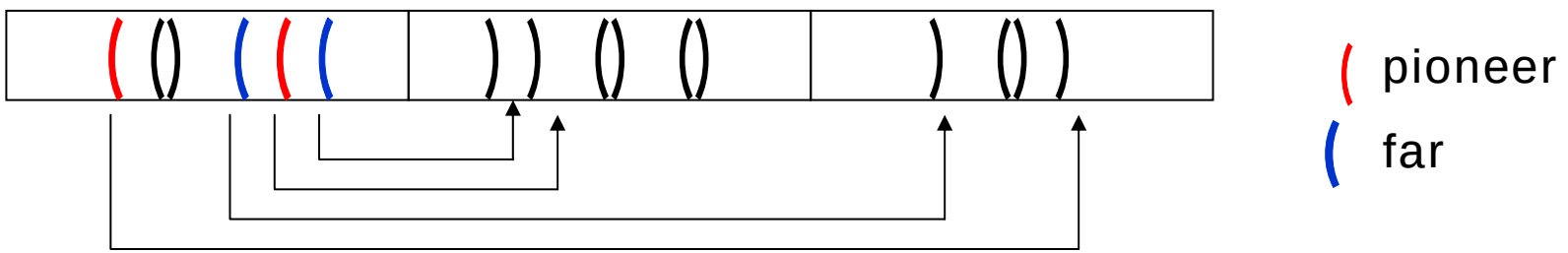


((()(()())))

0010010111

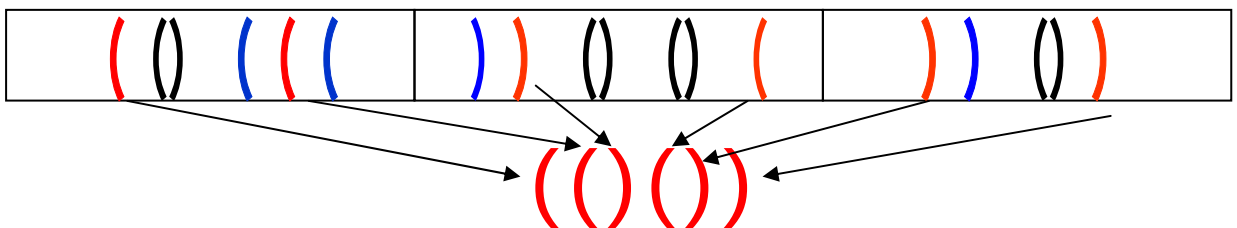
PTの実装

- 括弧列をサイズMのブロックに分割
- 括弧をこの分割に従って次のように分類
  - *near* “対応する括弧が同一ブロック内に存在”
  - *far* “*near*ではない”
  - *pioneer* “*far*であり、かつ対応する括弧のブロックが直前のfarの対応する括弧が存在するブロックとは違うブロックに存在”



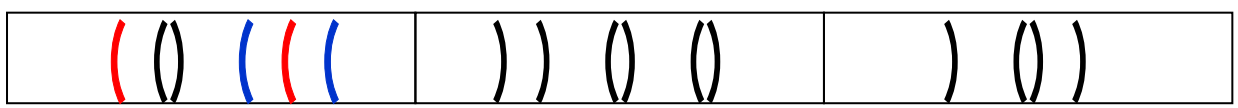
PTの実装 (続)

- **対応する括弧対が*pioneer*ならばその括弧もpioneerと定義する**
  - pioneer**のみから作った括弧列から再帰的に**PT**を作る**
- **定理**
  - Block**数が**B**の時、**pioneer**の数は**4B-6**以下**
  - **証明**: pioneer**の括弧対を結んだグラフは交差が無い平面グラフ。**



findcloseの例

- findclose(i)
- **i の種類による場合分け**
  - near
    - **前計算して作ったテーブル参照**
  - pioneer
    - pioneer**のみから作った**PT**の答えを参照**
  - far
    1. **直前の**pioneer**,** p**を探す** (rank**を利用**)
    2. p**までの開括弧の数 – 閉括弧の数を**t**とする** (rank**を使う**)
    3. p**の対応する括弧対の位置から**i**の対応する括弧のブロックを求める**
    4. p**の位置と**t**から対応する括弧の位置をテーブル参照で求める**



今後の展望

より複雑なデータ構造に対する索引

- 木、グラフの（圧縮）索引
  - 完全二分木に対するSuffix Tree [Shibuya 1999]
  - xbw [Ferragina 2006 to appear]
    - ラベル付き木に対するBWT
    - 作業領域量はtree entropy。Sub-pathなどが高速に求められる
- 近似マッチングの索引
  - ゲノムアライメントではpattern hunterという穴空きシードを使ったものが高速に近似マッチングを探索可能
  - ゲノムアライメント、統計的機械翻訳などで需要有り

理論的な枠組み

- **Ψ**, BWT**に代わる表現は存在するか？**
- **他の可逆変換の存在は？**
- **理論的な限界の向上**
  - NH_kbit **の作業領域を用いて**occ, loc**を**O(m)**時間で実現可能か？**