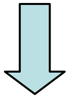
2006/07/05
第3回次世代アルゴリズム研究会＠大岡山

Succinct Data Structure

岡野原　大輔
東京大学情報理工学系研究科
コンピュータ科学専攻
hillbig＠is.s.u-tokyo.ac.jp

背景

- データ量の増加



- 保存･検索のコストが増大
- 計算コストの増大
  - 高速なメモリ内で計算できない
  - 一台に収まらないので並列化･通信が必要

目標

<u>データを小さく保持かつ高速な操作が可能なデータ構造</u>

発表の流れ

- 定義
- Succinct Data Structure (SDS)
  - ビット列に対するSDS
  - 木構造に対するSDS
  - 文字列に対するSDS
- SDSを用いた圧縮全文索引
  - Suffix ArraysとBurrow Wheelers 変換
  - FM-index, Compressed Suffix Arrays
- まとめ・今後の目標

発表の流れ

- **定義**
- Succinct Data Structure (SDS)
  - **ビット列に対する**SDS
  - **木構造に対する**SDS
  - **文字列に対する**SDS
- SDS**を用いた圧縮全文索引**
  - Suffix ArraysとBurrow Wheelers **変換**
  - FM-index, Compressed Suffix Arrays
- **まとめ・今後の目標**

定義(1/2)

- 計算モデル: word RAM
  - 入力長が*n*の時 log *n* ビットの操作は定数時間
  - 例　通常 n～$2^{32～64}$ であり64bit操作が定数時間
- 情報理論的下限
  - データ集合*D*の情報理論的下限*L*は
    $L = \log($*D*の場合の数$)$
  - 全ての場合が等確率で出現する場合に
    *D*の要素を保存可能な最小サイズ
  - 例　*n* 節点の順序木*T* の場合数は $\frac{1}{n+1}\binom{2n}{n} \approx 4^n$
    L　　$2n - \Theta(\log n)$

定義(2/2)
経験エントロピー [Manzini 01]

- H_0:0次経験*エントロピー
  - n_c : T中のcの出現回数

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{|T|} \log \frac{|T|}{n_c}$$

- H_k : k次経験エントロピー
  - T^s : $s \in \Sigma^k$の直前に出現した文字を連結

$$H_k(T) = \sum_{s \in \Sigma^k} \frac{|T^s|}{|T|} H_0(T^s)$$

- $H_k \leqq H_{k-1} \leqq \ldots \leqq H_1 \leqq H_0 \leqq 1$が成り立つ
- データ生成源のモデルは未知であってもよい

例

- T = *aacbbcbc*
- $|T| = 8$, $n_a = 2$, $n_b = 3$, $n_c = 3$
- k=0の場合
  - $H_0(T) = (2/8)*\log(8/2) + \ldots$ 0.47
- k=2の場合
  - $\Sigma^2 = \{ac, cb, bb, bc, c\$, \$\$\}$
  - $T^{ac} = a$ $T^{cb} = ab$ $T^{bb} = c$ …
  - $H_2(T) = (1/8)*0 + (2/8)*1 + \ldots + \ldots$ 0.25

- 実データのH_5の例　英文:0.23　DNA:0.24　XML:0.10

発表の流れ

- 定義
- Succinct Data Structure (SDS)
  - ビット列に対するSDS
  - 木構造に対するSDS
  - 文字列に対するSDS
- SDSを用いた圧縮全文索引
  - Suffix ArraysとBurrow Wheelers 変換
  - FM-index, Compressed Suffix Arrays
- まとめ･今後の目標

Succinct Data Structure (SDS)とは

- あるデータ集合*D* を格納するデータ構造
  - データ集合の例 ビット列、木、グラフ、文字列
- データ構造のサイズ: 情報理論的下限に近い
  - 情報理論的下限 $L = \log$ (*D* の場合の数)
  - 補助データ構造を利用して (1+o(1))L bits
- 高速な問合せ
  - 展開せずにO(1)もしくはo(L)時間でデータ構造中の列挙・探索が可能
  - 補助データ構造(索引) を利用。索引サイズは漸近的に無視できる

ビット列(集合)に対するSDS

- ビット列 B[0…n-1]: B[i]=1 または 0
  - 集合D⊂{0…n-1}を表現するには
    i∈D ならば B[i]=1,そうでないならB[i]=0
- 次の問い合わせをサポート
  - *lookup*(B,*i*): B[i]を返す
  - $rank_1(B,i)$: B[0…i]中の1の数を返す
  - $select_1$(B,*j*):B中の(j+1)番目の1の位置を返す

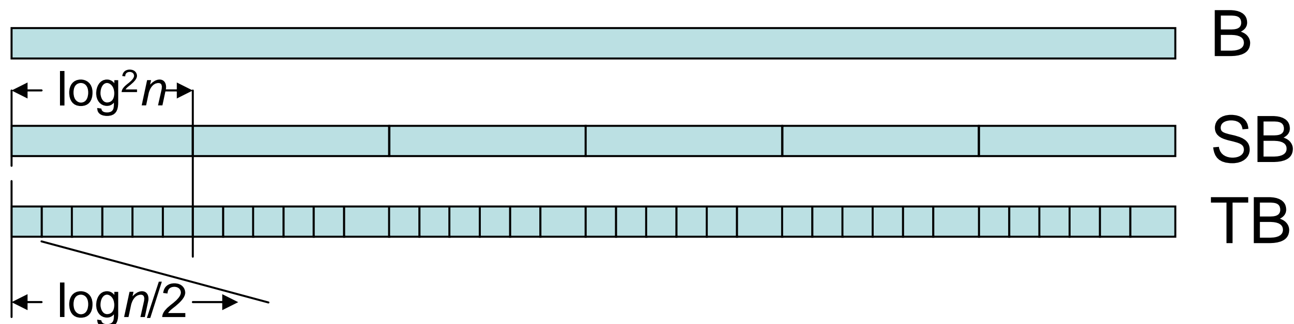
B[0…20] = *000100101010010010010*
$lookup_1$ (B,*0*)=*0*, $lookup_1$ (B,*6*)=*1*
$rank_1$ (B,*10*)=*4*, $rank_1$ (B,*15*)=*5*
$select_1$ (B,*0*)=*3*, $select_1$ (B,*4*)=*13*

ビット列(集合)に対するSDSの実現

- 基本的な考え
  - 再帰的に小さいブロックに分割
  - 大きなブロックの結果は明示的に保存
  - 一番小さいブロックは表引き(c.f. word RAM モデル)
- 様々な方法が提案
  - n+o(n) bit [Jacobson 89] [M96]
  - H_0(B)+o(n) bit [Grossi 02]
  - これらは実装が難しい
- 実用的な方法も提案されている [Gonzalez 05] [Kim 05]

ビット列(集合)に対するSDSの実現 rankの例

- Bを長さ$\log^2 n$毎のブロックに分割 (SB: Super-Block)
- 各SBを長さ$\log n/2$毎のブロックに分割 (TB:Tiny-Block)
- 各SBの先頭のrankの結果を保持 O(n/logn) bit
- 各TBの先頭のrankと直前のSBの先頭とのrankの差 (相対rank) を保持 O(n loglogn / logn) bit
- TB内のrankは表引き、もしくはpopCount (次ページ)
- rank(B,i) = SB[i/log2n]+TB[i/logn*2] + rank(残りのrank)



popcount(x)

x中の1の数を数える

```
unsinged int popCount(unsinged int r) {
  r = ((r & 0xAAAAAAAA) >> 1) + (r & 0x55555555);
  r = ((r & 0xCCCCCCCC) >> 2) + (r & 0x33333333);
  r = ((r >> 4) + r) & 0x0F0F0F0F;
  r = (r>>8) + r;
  return ((r>>16) + r) & 0x3F;
}
```

0xAAAAAAAA = $1010101010...10_2$

0x55555555 = $0101010101...01_2$

0xCCCCCCCC = $1100110011...00_2$

0x33333333 = $0011001100...11_2$

0x0F0F0F0F = $0000111100...11_2$

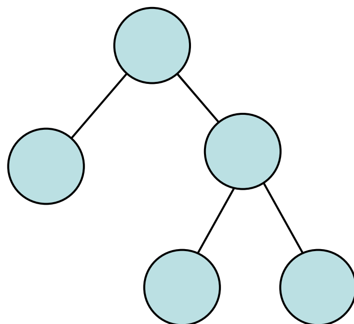
ビット列(集合)に対するSDSの実現 selectの例

- rankの二分探索　O(log n)時間
  - $select_1(B,j)$: $rank_1(B,k) < j \leqq rank_1(B,k+1)$ となるk
- o(n)作業領域で定数時間の実現は複雑
  - 1と1の間隔が最悪nになる可能性
- Algorithm I [Kim 2005]
  - Bを長さ$\log^{1/2} n$のブロックに分割
  - 1を一つも含まないブロックを除く
  - $\log^2 n$ずつの1の位置を明示的に記録
  - $\log^{1/2} n$ずつの1の相対位置を明示的に記録 (1と1の間隔が最悪$2\log^{1/2}$で抑えられるので可能)
- 1が少ない場合、明示的に答えを保存可能

木に対するSDS

[Jacobson 85] [Munro 01] [Geary 05] [Benoit 05]

- **節点数nの順序付き木を**2n bit**で表現**
  - **従来のポインタ表現は**O(nlogn) bit
    (**親、最初の子、次の兄弟へのポインタで**96n bit)
- Balanced Parenthesis (BP) [Munro 01] [Geary 05]
  - **木を**DFS**で辿り、節点に入る時**’(‘、**出る時**’)’**を記録**
- Depth First Unary Degree Sequence (DFUDS)
  - **最初に**’(‘。**木を**DFS**で辿り、各節点で、子の数が**k**の節点を**
    k**個の**’(‘**と**1**つの**’)’**を記録**



BP	DUFDS
(()(()()))	((())(()))
0010010111	0001100111

BPがサポートする操作

次の操作をchild,childrank以外定数時間*でサポート

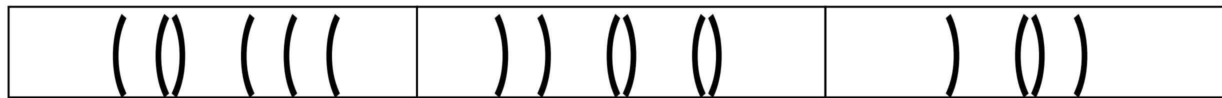
- *parent* (*x*) xの親
- *firstchild* (*x*)最初の子
- *sibling*(*x*):次の兄弟
- *depth*(*x*): xの深さ (根までの節点数)
- *desc*(*x*): xの子孫数
- *rank*(*x*): xのpreorderの順位
- *select*(*i*): preorderでiの順位を持つ節点
- *LA*(*x, d*): xの祖先で、深さdの節点 (*level-ancestor*)
- *lca*(*x, y*): xとyの最小共通祖先
- *degree*(*x*): xの子供の数
- *child* (*x, i*): xのi番目の子
- *childrank* (*x*): xの親からみて、xは何番目の子か

今回説明するのはこれら

*DFUDSはlca, depth, LA以外定数時間

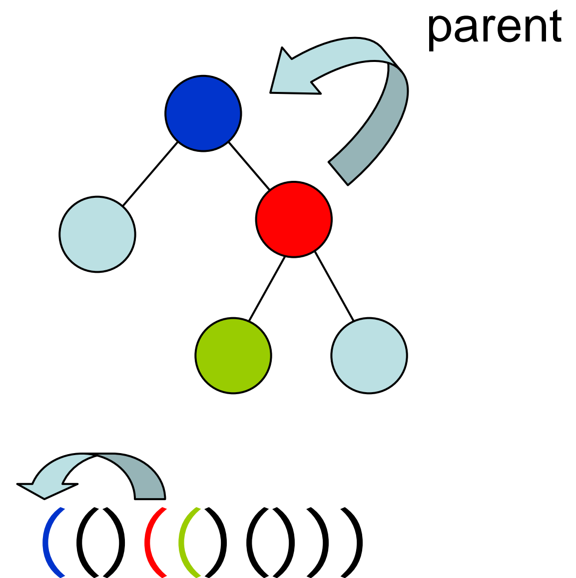
BPの定義

- n個の節点,葉を持つ順序木からビット列B[0…2n-1]を次のように構築
  - 木をDFSで辿り、節点に入る時'('、出る時')'
  - '('の場所で節点、葉情報を保持。B上の$rank_($、$select_($ でBP中の位置と節点情報を対応付ける
- Bを長さM=log*n*/2のブロック$B_0 \ldots B_{(2n-1)/M}$に分割
- m(x) : xの対応する括弧の位置
- b(x) : xが所属するブロック番号



BPでの操作の実現

- BP上で次の操作をサポート (次ページで説明)
  - findopen(*x*), findclose(*x*): xに対応する括弧対の位置を返す
  - enclose(*x*): xを最もきつく囲む括弧対の開括弧の位置を返す
- 上記操作で木の操作を実現
  - parent(x) = enclose(x)
  - sibling(x) = findclose(x)+1
  - first-child(x) = x+1



findcloseの実装(1/4) [Munro 01] [Geary 05]

- 各括弧x を次のように分類
  - *x*が*near* ⇔ b(x) = b(m(x))
    “対応する括弧が同一ブロック内に存在”
  - *x*が*far* ⇔ b(x) ≠ b(m(x))
- findclose(x)（失敗バージョン）
  - xがnearなら表引き
    表の大きさは$O(n^{1/2}(\log n)^2) = o(n)$
  - xがfarの場合も表引き?
    - 全ての括弧がfarの時、サイズが$n\log n$となり不適

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

全てがfarの場合

findcloseの実装(2/4)

- ***解決策***: *far*の全てを記録しなくても良い
  - *x*が*pioneer* : xが*far*、かつxの直前のfarである括弧をx'とした時、b(m(x))≠b(m(x'))
  - またm(x)が*pioneer*ならばxも*pioneer*とする
- **定理**:
  Block数がTの時、pioneerの数は4T-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　オイラーの定理を適用

(pioneer
(far

findcloseの実装(3/4)

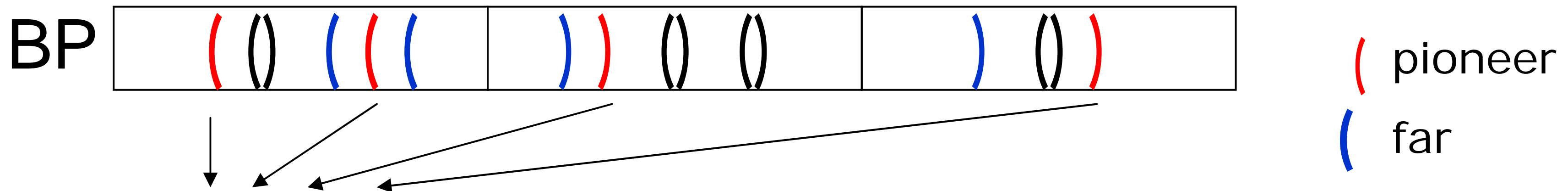
- Pioneerである括弧のみから括弧列BP_2を作る
  - BP_2はバランスがとれており、長さはO(n/logn)
  - BP_2 でもBPと同様に操作を行う (再帰的)
- BPとBP_2を対応付けるビット列P[0…2n-1]を用意
  - B[i]がPioneerならばP[i]=1 それ以外P[i]=0
  - P上のrank, selectでBP中のpioneerとBP_2中の括弧を対応付け

findcloseの実装(4/4)

- findclose(x)（正式バージョン）
  - xがnearの場合　表引き
  - xがpioneerの場合　BP_2の結果を使う。
  - xがfarの場合
    (1)直前のfar x’ をselect(rank(P,x))で求める
    (2) y = μ(x’)を求め、,B(y)を求める。
    　(B(y) とB(μ(x))は同じ)
    (3)xとx’の間の’(‘の数と’)’の数の差を調べ,
    　B(μ(x))中の対応する括弧を表引き
- findcloseも同様。encloseはちょっと複雑

BPの例

P　100010　010000　0001



BP_2　(())

<u>findclose(4)</u>
B[4]はpioneerなのでBP2の対応する位置(1=rank(P,4)-1)でのfindcloseの結果を求めて、2を得て、select(P,2)= 7

<u>findclose(3)</u>
B[3]はfarなので直前のpioneerの位置を求め、その対応する括弧の位置を求める。

文字列に対するSDS

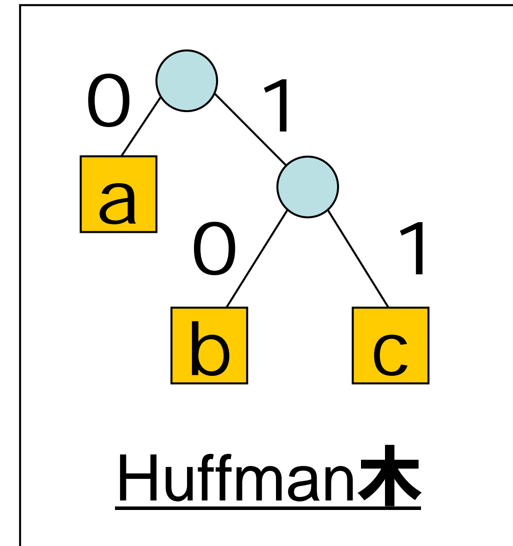
- T[0…n-1]、T[i]∈Σが与えられた時、c∈Σに対し $rank_c(T,i)$ や $select_c(T,i)$ をサポート
  - これらは、圧縮全文索引に必要
- |Σ|=2の場合→ビット列の場合で説明済
- |Σ|>2の場合
  - ビット列に対するSDSと同様にSB,TBに分けそれぞれでの答えを記録する方法をとるとサイズは O(|Σ| *(n/logn + n loglog n/ logn))
  - |Σ|>lognの時、補助データ構造のサイズはnより大きい（一般データでは|Σ|=256…65536 log*n*<32）

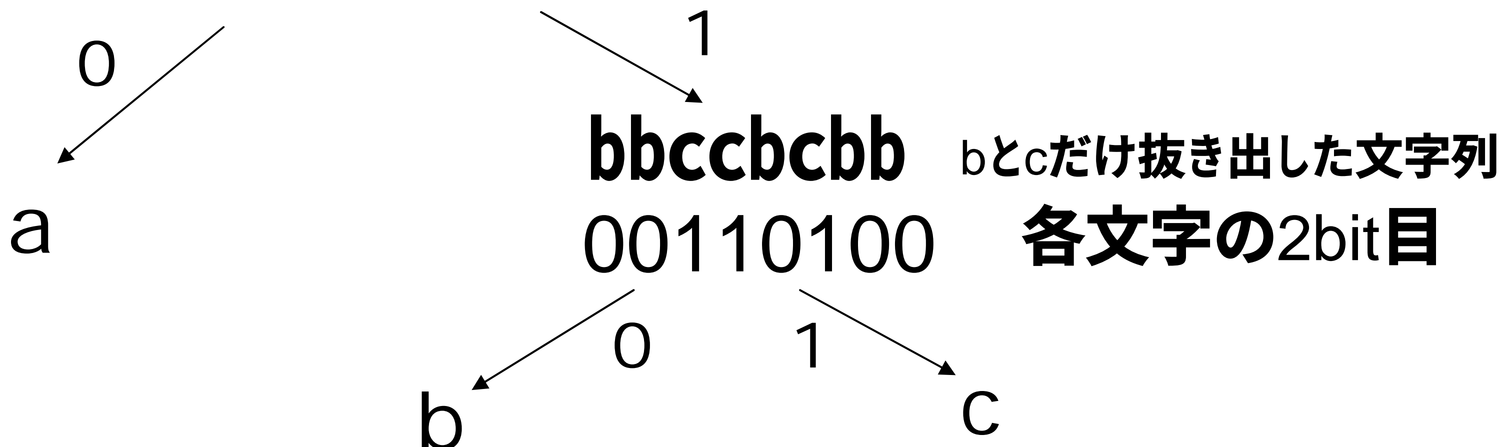
文字列に対するSDS

- 様々な手法が提案
  - |Σ|<o(n/loglogn)の時、Generalized Wavelet Tree が最小、最速 [Ferragina 2004]
  - $nH_0(T)$ bits で rank, selectを定数時間
- 現実的にはWavelet Treeが有効 [Grossi 2003]
  - $nH_0(T)$ bits で rank, selectをlog(Σ)時間
  - 各節点にビット列が付随したHuffman木
    - 葉には各文字が対応
    - rankは根から葉、selectは葉から根へそれぞれrank,selectを適用する。

Waveletの例(1/2)

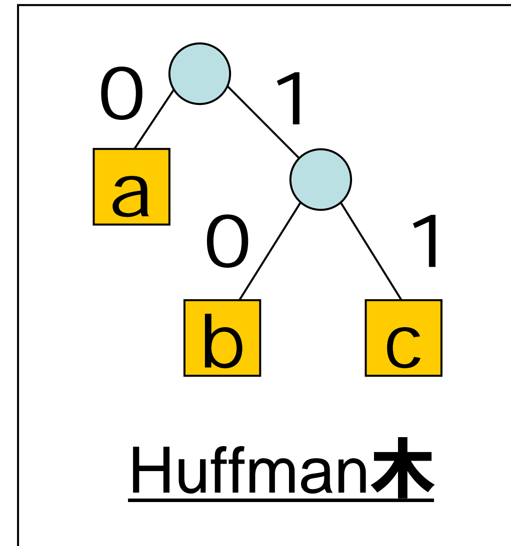
- Σ={a,b,c}
  $a = 0_2$ $b = 10_2$ $c = 11_2$
- T= abbccbaacbab





Wavelet Treeの例(2/2)

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



abbccbaacbab
011111001101

rank$_1$(8)=5

0 1

a

bbccbcbb
00110100

rank$_b$(T,8)=3

rank$_0$(T,5)=3

0 1

b c

発表の流れ

- 定義
- Succinct Data Structure (SDS)
  - ビット列に対するSDS
  - 木構造に対するSDS
  - 文字列に対するSDS
- SDSを用いた圧縮全文索引
  - Suffix ArraysとBurrow Wheelers 変換
  - FM-index, Compressed Suffix Arrays
- 今後の流れ

（圧縮）全文索引の問題設定/用語

- 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)
  - パタンP (長さm)
  - occ(P) パタンPのT中の出現回数
  - loc(P) パタンPのT中の全ての出現位置
- 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（本発表では文字索引のみを扱う）
  - 任意の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
  - 明示的に情報を保持しない場合が多い
- 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

文字索引

- 文字索引：任意のパタンPに対して答えられる
- 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転置ファイルが最近利用されている
  - 長さnの全部分文字列の出現を記録
  - 辞書ヒット数が大きい場合、計算量はO(N)に近づくため、大規模データへの適用は困難

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、作業領域量は大きくなる**
- **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**
- **サイズと速度のトレードオフ**

  →作業領域量、速度の両面で*ほぼ*最適なものが達成可能
  - Suffix Arrays (BW**変換**)**と**SDS**を組み合わせる**
  - **最小・最速** nH_kbit**で**occ(P)**を**O(m) **時間**
    [Ferragina 2005]

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**

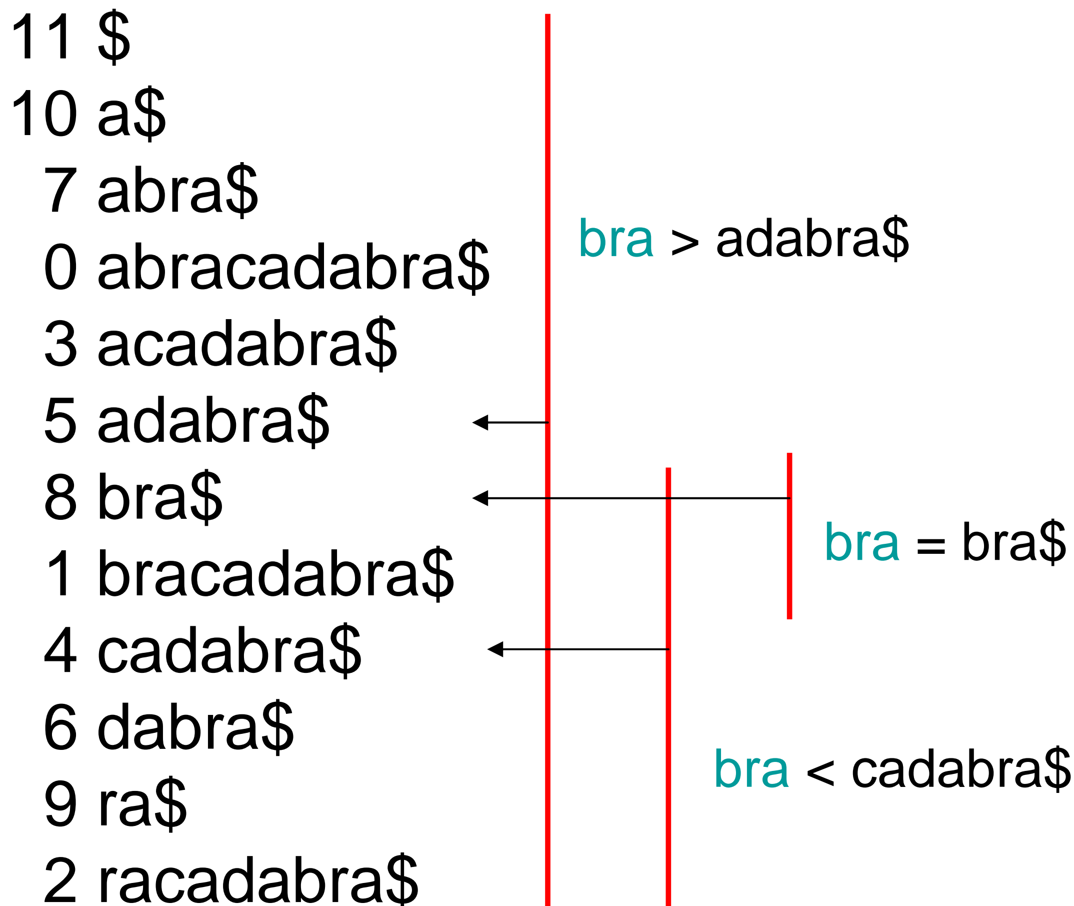
(3) **接尾辞の番号を抽出**

(2) **接尾辞集合を辞書式順序でソートする**

SAを使った検索

入力 T=abracadabra$ **パタン** P = bra

二分探索を行う



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 (5n byte)

Hgt**配列を使うと**
occ(P)**は**$O(m+\log n)$

Compressed Suffix Arrays (CSA)

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

- SAと同じ操作を元テキストと同じもしくは小さい作業領域量で実現
  - 元のSAの作業領域量は $n \log n$ bit。SAは0からN-1までの並び替えで圧縮しにくい
- SAの代わりに次のように定義されるΨを保存
  - $\Psi[i] = SA^{-1}[SA[i] + 1]$
  - 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
  - $SA[i] = SA[\Psi[i]]-1 = SA[\Psi^2[i]]-2 = \ldots = SA[\Psi^n[i]]-n$
- もし $SA[\Psi^n[i]] = p$ ならば $SA[i] = p-n$

Ψの性質

- **Ψは**SA**とは違い圧縮しやすい**
- **<u>定理</u>**　T[SA[i]] = T[SA[i+1]] **ならば** **Ψ**[i] < **Ψ**[i+1]
- **<u>証明</u>**　T[SA[i]] = T[SA[i+1]]**ならば**SA[i]**と**SA[i+1]**の順位は辞書式順序の定義より、それらより一文字短い**Suffix**の順位で決まる。よって、**
  $SA^{-1}[SA[i]+1] < SA^{-1}[SA[i+1]+1] \Rightarrow \Psi[i] < \Psi[i+1]$
- d[i]:=**Ψ**[i+1]-**Ψ**[i]**をδ符号で保存した場合サイズは**nH_0 bits (**実際は**nH_kbits**に近い**) [Sadakane 2003]
- d**を**wavelet tree**で保存するとサイズは**nH_kbit [Grossi 2003]

abra$　　　　bra$

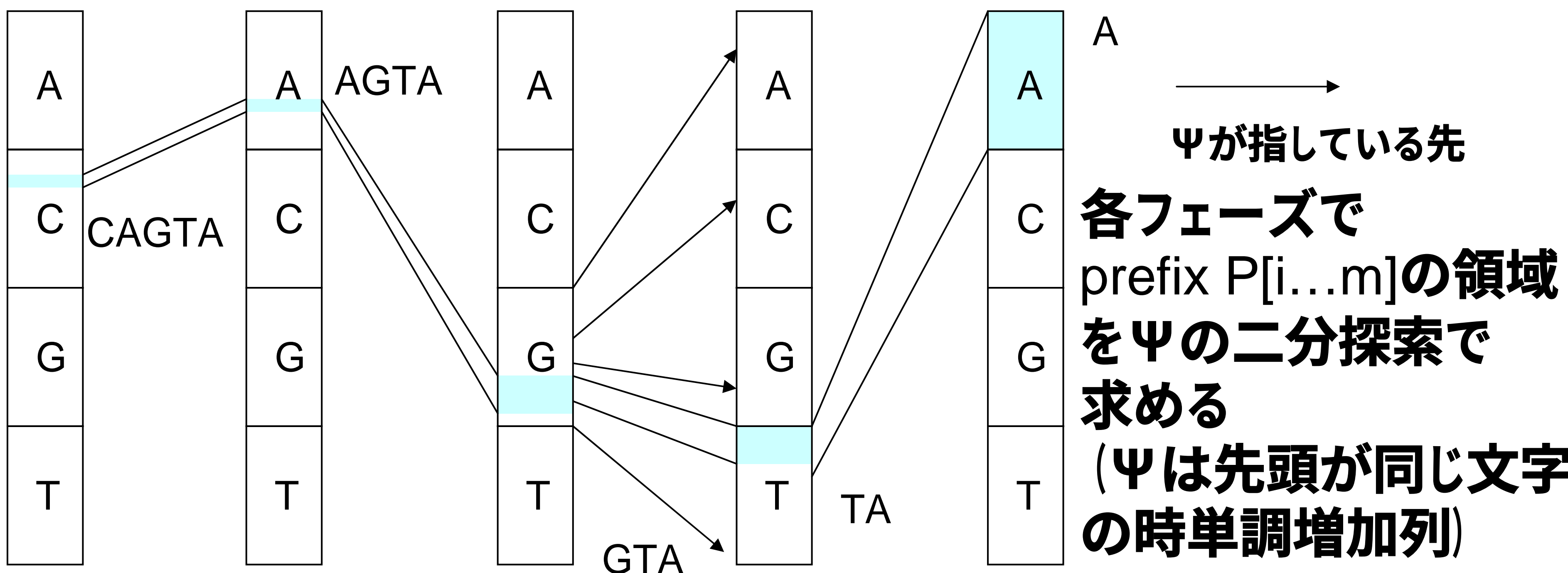
abracadabra$　bracadabra$

↑ **1文字目が同じ場合**

↑ **2文字目以降** (SA[i]+1**と**SA[i+1]+1) **から始まる部分列で順位が決定される。**

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]…)



Burrows Wheeler's Transform [1994] (BWT)

- **文字列に対する可逆変換（並び替え）**
- **定義** BWT[i] := T[SA[i]-1]
  - **但し**SA[i]=0**の時** BWT[i] = T[n]
- **例** abracadabra$ ⇒$_{BWT}$ ard$rcaaaabb
- BWT**後の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**
  - c.f. Compression boosting [Ferragina 2005]

t hese are possible ...
t hese were not of ..
t hese ...

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 …..**続く**

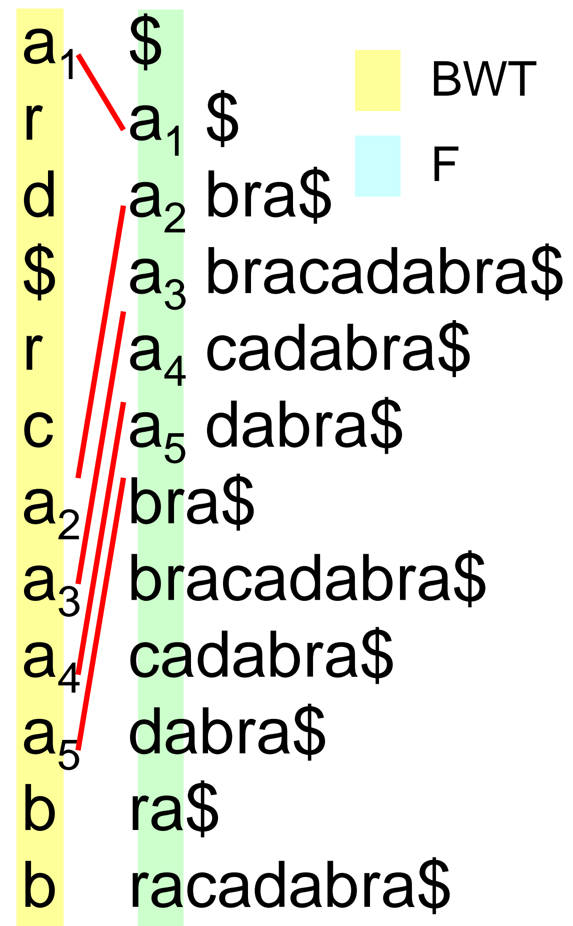
BW**変換**

BWT後

Iooooioororooorooooooooooooroooorromrrooomooroooooooooormoorooororioooro
ormmmmmuuiiiiiIiuuuuuuuuiiiUiiiiiioooooooooooooroooooiiiioooioiiiiiiiiiioiiiiiieuiiiiiiiii
iiiiiiiiouuuuouuUUuuuuuuooouuiooriiiriirriiiiriiiiiiaiiiiiooooooooooooooiiiouioiiiiioiiuii
uiiiiiiiiiiiiiiiiiiiiioiiiiioiuiiiiiiiiiiiioiiiiiiiiiiiioiiiiiuiiiioiiiiiiiiiiiioiiiiiiiiiioiiiioiiiiiiioiiiaiiiiiiiiiiiiiiiioii
iiiioiiiiiiiiiiiiiiuiiiiiiiiiiiiiiiiioiiiiiiioiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiuuuiioiiiiiuiiiiiiiiiiiiiiiiiiiiiiioiiiiuioiuiiii
iiioiiiiiiuiiiiiiiiiiiiiiiiiiiiiiiiiiiiioaoiiiiioioiiiiiiiioooiiiiiooioiiioiiiiiouiiiiiiiiiiiooiiiiiiiiiiiiiiiiiiiiiiiiiiii
ioiiiiiiiiiiiiiiiiiioiooiiiiiiiiiiioiiiiiuiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioiiiiiiiiiiiioiiiuiiiiiiiiiioiiiiiiiiiiiuoiiioiiioiii
iiiiiiiiiiiiiiiiiiuiiiiuuiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiuiuiiiiiuuiiiiiiiiiiiiiiiiiiiiiiiuiiiiiiiiiiiiiiiiiiiiiiiiiiioiiiiiiioiiiiiii
iiiiiiiiiiiiioiiiiiiiiioiiiiuiiiioiiiioiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioiioiiiiiiuiiiiiiiiiiiiiiooiiiiiiiiiiiiiiiiiiiioooiiiiii
iioiiiiouiiiiiiiiiiiiiii
…..**続く**

BWTの重要な性質

- F: 各Suffixの先頭文字をつなげたもの
- <u>定理</u>
  - BWTの任意の文字cについて上から順に番号を付けた時、それらに対応する文字はFでも同じ順に出現する
- <u>証明</u>
  - BWT, Fの各文字の順序はそれに続く文字列の順位で決まるが、それに使われる文字列はどちらも同じ文字列



BWTの重要な性質 (続)

- SA^{-1}: SAの逆関数 SA[k]=i の時 $SA^{-1}[i]=k$
- cum[c] := T中のcより小さい文字の個数
- これらを使って先程の定理を言い換えると
  - $SA^{-1}[SA[i]-1] = rank_c(BWT,i-1) + cum[c]$
  - $SA^{-1}[SA[i]+1] = select_x(BWT,i-cum[x])$
    ただし、c=BWT[i]
    xは$cum[x] \leqq i \leqq cum[x+1]$を満たすx
- BWTの逆変換は後者(lf-mapping)を利用する

i	SA	SA^{-1}	$SA^{-1}[SA[i]+1]$	BWT	Suffix
0	11	3	3	a	$
1	10	7	0	r	a$
2	7	11	6	d	abra$
3	0	4	7	$	abracadabra$
4	3	8	8	r	acadabra$
5	5	5	9	c	adabra$
6	8	9	10	a	bra$
7	1	2	11	a	bracadabra$
8	4	6	5	a	cadabra$
9	6	10	2	a	dabra$
10	9	1	1	b	ra$
11	2	0	4	b	racadabra$

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,'$'); //return the position of '$'
    for (int i = 0; i < n; i++){
        next = LFmapping[next];
        putchar(bwt[next]);
    }
    delete[] LFmapping;
}
```

FM-index [Ferragina 2000]

- BWTを基にしたもう一つの圧縮全文索引*
  - BWT[i] := T[SA[i]-1] を保存・利用
- BWT上のrank, select操作でSAを実現
  - $SA^{-1}[SA[i]-1] = rank(BWT,c) + cum[c]$
  - $SA^{-1}[SA[i]+1] = select(BWT,c)$
- CSAと同様に出現頻度、出現場所、部分文字列復元などをサポート

*圧縮全文索引は他にLZ-index [Karkkainen 96]が有名

FM-indexの検索

入力	P[0…m-1]:**検索したいパタン**
	BWT[0…n-1]: **検索対象テキストTのBWT後のテキスト**
	C[0… Σ -1]: C[c]**はcより小さい文字がBWTに現れた回数を保持**
返り値	[sp,ep] P**をprefixとして持つsuffix arraysの範囲。**
	ep<sp**ならば、PはT中に存在しなかったと報告**

```
1.  i := m-1
2.  sp := 0; ep := n-1;
3.  while (sp ≦ ep) and (i >= 0) 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.  end
```

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

P="abr"
T="abracadabra$"
BWT="ard$rcaaaabb"

sp := 0 ep := 11	
sp := 9+0+1 = 10 ep := 9+2 = 11	i = 2 c='r'
sp := 5+0+1 = 6 ep := 5+2 = 7	i = 1 c='b'
sp := 1+1+1 = 3 ep := 1+3 = 4	i = 0 c='a'

```
i := m-1
sp := 0; ep := n-1;
while (sp ≦ ep) and (i >= 0) do
    c := P[i];
    sp := C[c]+rank(BWT,c,sp-1)+1;
    ep := C[c]+rank(BWT,c,ep);
    i--;
end
```

発表の流れ

- 定義
- Succinct Data Structure (SDS)
  - ビット列に対するSDS
  - 木構造に対するSDS
  - 文字列に対するSDS
- SDSを用いた圧縮全文索引
  - Suffix ArraysとBurrow Wheelers 変換
  - FM-index, Compressed Suffix Arrays
- まとめ･今後の目標

まとめ

- Succinct Data Structures (SDS)
  - 小さいサイズ (情報理論的下限) で高速 (定数時間) の操作が可能
  - 複雑なデータ構造もビット列の操作に還元
- 圧縮全文索引
  - Suffix Arrays、Burrows Wheeler 変換で検索問題を二分探索、rank、select問題に還元
  - SDSの利用でデータサイズがnH_k bitsを達成

今後の目標

- Succinct Data Structures
  - 複雑なデータ構造への対応
    - 木、グラフ、関数、行列、多次元情報
  - より小さく、速いデータ構造
    - nH_kbits O(1)時間
    - 経験エントロピー以外のサイズ (Gap Measure [Gupta 06])
- 圧縮全文索引について
  - より小さく、速いデータ構造 (外部記憶領域との親和)
  - 複雑なクエリーのサポート
    - 近似マッチング[Huynh 2005], 正規表現, 文書頻度
- 動的なデータ構造
  - log*n*時間で挿入、削除可能なビット列SDS [Makinen 06]