算法数理工学
第12回

定兼　邦彦

動的計画法
(Dynamic Programming)

- 分割統治法と同様，部分問題の解を統合することによって問題を解く.
- 分割統治法では問題を独立な部分問題に分割し，部分問題を再帰的に解き，それらの解を組み合わせて元の問題の解を得る.
- 動的計画法では部分問題が独立でないときに用い，計算量を削減する.

動的計画アルゴリズムの開発

1. 最適解の構造を特徴付ける
2. 最適解の値を再帰的に定義する
3. ボトムアップ方式で最適解の値を計算する
4. 計算された情報からある最適解を構成する

文字列の編集距離

- 文字列 X と Y の編集距離 (edit distance) とは, X に以下の編集操作を繰り返して Y に変換する際の最小の操作回数である.
  - 挿入: x_i と x_{i+1} の間に文字 c を入れる
  - 削除: x_i を削除
  - 置換: x_i を文字 c で置き換える

X = ACGTT

↓ A を削除

CGTT

↓ C を挿入

CGCTT

↓ T を A に置換

Y = CGCAT

編集距離 = 3

- X の文字を削除する代わりに，Y に gap (空白) を入れる
- X に文字を挿入するときは必ず Y の文字を入れることになる ⇒ X に gap を入れる
- X と Y に gap を入れた X' と Y' で，ミスマッチの数を最小にする問題と等価

X = ACGTT ⟶ X' = ACG–TT　　ミスマッチ = 3

Y = CGCAT ⟶ Y' = –CGCAT

- $X_i = <x_1, x_2, \ldots, x_i>$ と $Y_j = <y_1, y_2, \ldots, y_j>$ の編集距離を $c[i,j]$ とする
- x_i と y_j をマッチさせる場合
  - $x_i = y_j$ ならば $c[i,j] = \mathrm{c}[i-1,j-1]$
  - $x_i \neq y_j$ ならば $c[i,j] = \mathrm{c}[i-1,j-1]+1$
- x_i の次に gap を入れ y_j とマッチさせる場合
  - $c[i,j] = c[i,j-1]+1$
- y_j の次に gap を入れ x_i とマッチさせる場合
  - $c[i,j] = c[i-1,j]+1$
- $c[i,j]$ はこれら３つの中の最小値
- O(mn) 時間 ($m = |X|, n= |Y|$)

- 「↑」は Y に gap を入れることを表す
- 「←」は X に gap を入れることを表す
- 斜めは一致または置換を表す

X = ACGTT ⟶ X' = ACG–TT

Y = CGCAT ⟶ Y' = –CGCAT

	j	0	1	2	3	4	5
i		y_j	C	G	C	A	T
0	x_i	0	← 1	← 2	← 3	← 4	← 5
1	A	↑ 1	↖ 1	↖ 2	↖ 3	↖ 4	↖ 5
2	C	↑ 2	↖ 1	↖ 2	↖ 3	↖ 4	↖ 5
3	G	↑ 3	↑ 2	↖ 1	← 2	← 3	← 4
4	T	↑ 4	↑ 3	↑ 2	↖ 2	↖ 3	↖ 3
5	T	↑ 5	↑ 4	↑ 3	↖ 3	↖ 3	↖ 3

連鎖行列積問題

- 入力: n 個の行列 $\langle A_1, A_2, \ldots, A_n \rangle$
  - 各行列の行数，列数は異なる
- 出力: 行列積 $A_1 A_2 \ldots A_n$ を最も速く計算する行列計算の順序
- 行列の積は結合的なので，計算順を変えても答えは同じ
- 行列 A が $p \times q$, 行列 B が $q \times r$ のとき，積 $C = AB$ は $p \times r$ で，計算時間は $\mathrm{O}(pqr)$ とする
- 計算順を変えると計算時間が変わる

- 入力が $\langle A_1, A_2, A_3, A_4 \rangle$ のとき，計算順序は
  - $(A_1\ (A_2\ (A_3\ A_4)))$
  - $(A_1\ ((A_2\ A_3)\ A_4))$
  - $((A_1\ A_2)(A_3\ A_4))$
  - $((A_1\ (A_2\ A_3))\ A_4)$
  - $(((A_1\ A_2)\ A_3)\ A_4)$

  の5通り
- 行列積の計算順は，括弧のつけ方で表現できる

異なる括弧付けの個数

- n 個の行列の異なる括弧付けの個数を $P(n)$ とする

$$P(n)=\begin{cases} 1 & n=1 \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & n\geq 2 \end{cases}$$

- $P(n)$ は Catalan 数と呼ばれ，$\Omega(4^n/n^{3/2})$ である．
つまり全ての括弧付けを試すと指数時間かかる．
- 部分構造最適性を見つける必要がある

最適な括弧付けの構造

- $A_{i..j} = A_i\, A_{i+1}\, \ldots\, A_j$ とする
- $A_{i..j}$ を計算するには，ある k $(i \le k < j)$ に対し $A_{i..k}$ と $A_{k+1..j}$ を計算し，それらの積を計算する
- $A_{i..j}$ を計算するコストは，$A_{i..k}$ と $A_{k+1..j}$ のコストとそれらの積のコストの和
- $A_i\, A_{i+1}\, \ldots\, A_j$ の最適括弧付けでは積を A_k と A_{k+1} の間で分割するとする．このとき，最適括弧付けの前半部分は，$A_i\, A_{i+1}\, \ldots\, A_k$ の最適括弧付けである
  - もし $A_i\, A_{i+1}\, \ldots\, A_k$ に対するより良い括弧付けがあるなら，$A_i\, A_{i+1}\, \ldots\, A_j$ の最適括弧付けにもより良いものが存在．
  - 注: $A_{i..k}$ と $A_{k+1..j}$ の積の計算コストは一定

再帰的な解

- $m[i,j]$ を $A_i\,A_{i+1}\,\ldots\,A_j$ の連鎖行列積問題での最適解でのスカラー乗算の数とする
- $m[i,j]$ は再帰的に定義できる

$$m[i,j]=\begin{cases} 0 & i=j \\ \min\limits_{i\le k<j}\left\{m[i,k]+m[k+1,j]+p_{i-1}p_k p_j\right\} & i<j \end{cases}$$

  – p_k はA_kの列数，p_0 は A_1 の行数とする
- 部分問題の個数は $\mathrm{O}(n^2)$
- $m[i,j]$ は $j–i+1$ 個の行列積問題の解で，その計算で用いる m は $j–i$ 個以下の行列積問題の解
- 部分問題を長さの短い順に解けばいい

Matrix-Chain-Order(p)

時間計算量 O(n^3)
領域計算量 O(n^2)

for $i = 1$ to n $m[i,i] \leftarrow 0$
for $l = 2$ to n
  for $i = 1$ to $n-l+1$
    $j \leftarrow i+l-1$
    $m[i,j] \leftarrow \infty$
    for $k = i$ to $j-1$
      $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}\, p_k\, p_j$
      if $q < m[i,j]$
        $m[i,j] \leftarrow q$
        $s[i,j] \leftarrow k$
return m, s

動的計画法の基本要素

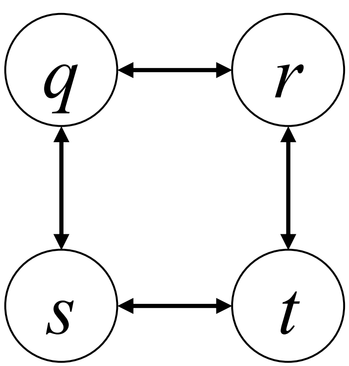
- 動的計画法がうまく働くために必要な性質

1. 部分構造最適性
2. 部分問題重複性

部分構造最適性

- ある問題が部分構造最適性を持つとは，問題の最適解がその内部に部分問題に対する最適解を含んでいることとする．
- 部分問題の最適解から，全体の最適解が効率的に求まる必要がある．
- 分割統治法でも同じ性質を使う．

- 重み無し最短路問題
  - 辺数最小の u–v パスを求める問題 (最適解は閉路を含まない)
- 重み無し最長路問題
  - 辺数最大の単純 u–v パスを求める問題 (閉路を含んでもいいならいくらでも長くなる)
- 最短路問題は部分構造最適性を持つ
- 最長路問題は部分構造最適性を持たない
  - 最長路問題はNP困難



$q \to r \to t$ は q から t への最長路だが，
$q \to r$ は q から r への最長路ではない．
q から r への最長路と r から t への最長路は
同じ辺を使う⇒繋げると閉路ができる

部分問題重複性

- ある問題を解くときに現れる部分問題に,
  同じ問題が現れること
- 部分問題が全て異なる場合は分割統治法で良い
- 部分問題の解を表に格納しておくことで,
  再計算を防ぐ

Recursive-Matrix-Chain(p, i, j)

時間計算量 $\Omega(2^n)$

1. if $i = j$ return 0
2. $m \leftarrow \infty$ // $m[i,j]$の値
3. for $k = i$ to $j-1$
4. $q \leftarrow$ Recursive-Matrix-Chain(p, i, k) + Recursive-Matrix-Chain(p, k+1, j) + $p_{i-1}\, p_k\, p_j$
5. if $q < m$ then $m \leftarrow q$
6. return m

動的計画法に基づくアルゴリズムでは，部分問題重複性を用いて計算量を落としている

時間計算量 $O(n^3)$
領域計算量 $O(n^2)$

Lookup-Chain(p, i, j)

if $i = j$ return 0
if $m[i,j] < \infty$ return $m[i,j]$
for $k = i$ to $j-1$
    $q \leftarrow$ Lookup-Chain(p, i, k)
        + Lookup-Chain(p, k+1, j) + $p_{i-1}\, p_k\, p_j$
    if $q < m[i,j]$ then $m[i,j] \leftarrow q$
return $m[i,j]$

Memoized-Matrix-Chain(p)

for $i = 1$ to n
    for $j = 1$ to n
        $m[i,j] \leftarrow \infty$
return Lookup-Chain(p, 1, n)

メモ化 (Memoization)

- 既に行った計算結果を表に格納しておき，2回目からは再計算せずにその表の値を返す
- 動的計画法と同じ計算時間を，再帰アルゴリズムで達成できる
- 一般には，表の全ての値が必要では無いので，データ構造として単なる配列を用いると無駄な場合がある
- その場合はハッシュ表などを用いて計算結果を管理する (動的計画法より計算量が増える)

文字列検索

- 情報検索で必須の処理
  - サーチエンジン
  - ゲノム情報処理
- データ量が莫大
  - Web: >20億ページ, 数テラバイト
  - DNA配列: ヒト=28億文字, 総計>150億文字

文字列検索問題

- 文字列 T の i 番目の文字を $T[i]$, i 番目から j 番目の文字からなる文字列を $T[i..j]$ と表記する
- 文字列 T の長さを $|T|$ と書く
- 文字列 P が, ある i と j に対して $P = T[i..j]$ となっているとき, P は T の部分文字列であるという. また, T の i 文字目は P とマッチするという
- 文字列検索問題は, 文字列 P と文字列 T を入力し, P が T の部分文字列となっている場所, つまり $P = T[i..j]$ となる i を全て見つける問題

文字列検索アルゴリズム

- 逐次検索
  - P と T が与えられてから問題を解く
  - 絶対に $O(|P|+|T|)$ 時間かかる
  - KMP法，BM法，Z法など
- 索引検索
  - T が予め与えられたとき，何らかのデータ構造 D を作っておく. P が与えられたときに D を用いて問題を高速に解く
  - 転置ファイル，接尾辞木，接尾辞配列など

逐次検索アルゴリズム

- 文字列マッチングを簡単に解こうと思ったら,
  各 i について $P = T[i..i+|P|-1]$ となるかどうかを
  調べればよい
- O($|T| \cdot |P|$) 時間のアルゴリズムができる

- もう一工夫して速くする方法を考える
- Z アルゴリズム
  - 最も単純な線形時間 (O($|T|+|P|$)) アルゴリズム

- 定義: 文字列 S と位置 $i > 1$ に対し, $Z_i(S)$ を S の i 文字目から始まる部分文字列で, S の接頭辞と一致するものの中で最長のものの長さと定義する.
- 例 : S = aabcaabxaaz のとき
  - $Z_2(S) = 1$ (aa…ab)
  - $Z_5(S) = 3$ (aabc…aabx)
  - $Z_6(S) = 1$ (aa…ab)
  - $Z_9(S) = 2$ (aab…aaz)
  - $Z_{10}(S) = 1$ (aa…az)
  - それ以外は $Z_i(S) = 0$

- 定義: $i > 1$ かつ $Z_i(S) > 0$ のとき, i でのZ-boxを区間 $[i, i+Z_i(S)-1]$ と定義する
- 定義: $i > 1$ に対し, r_i を $1 < j \leq i$ でのZ-boxの右端点の最大値と定義する. また, l_i をそのときの j と定義する. (j が複数ある時はどれでも可)
- 例: $S =$ a a b c a a b x a a z のとき

	1	2	3	4	5	6	7	8	9	10	11
S	a	a	b	c	a	a	b	x	a	a	z
Z_i		1	0	0	3	1	0	0	2	1	0
r_i		2	2	2	7	7	7	7	10	10	10
l_i		2	2	2	5	5	5	5	9	10	10

- Z は $O(|S|^2)$ 時間で計算できる
- $O(|S|)$ 時間で計算したい

- Z_i, r_i, l_i が $1 < i \le k-1$ に対して計算済みのとき，Z_k を計算する
- $r = r_{k-1}, l = l_{k-1}$ とする
- $k > r$ のとき (k がZ-boxに含まれないとき)
  - $S[k..n]$ と $S[1..n]$ を1文字ずつ比較して Z_k を求める
  - $Z_k > 0$ ならば $r = k+Z_k-1, l = k$
- $k \le r$ のとき (k があるZ-boxに含まれるとき)
  - $S[k] \in S[l..r]$
  - $S[l..r] = S[1..r-l+1]$ より，$S[k] = S[k-l+1]$
  - 同様に $S[k..r] = S[k-l+1.. r-l+1]$

- 2つの場合が考えられる
  - case 2a: Z_{k-l+1} が $S[k..r]$ の長さより小さいとき
    $Z_k = Z_{k-l+1}$

  - case 2b: Z_{k-l+1} が $S[k..r]$ の長さ以上のとき
    $S[r+1..n]$ と $S[r-k+1+1..n]$ を比較 (q 文字一致)
    $Z_k = (r-k+1)+q$
    $r = r + q$
    $l = k$

定理: 全ての Z_i は $\mathrm{O}(|S|)$ 時間で求まる

証明: ループの回数は $|S|$ 回.

文字列の比較は必ずミスマッチで終わる

⇒ミスマッチの回数はループの回数以下

マッチの回数を見積もる.

1回の文字列比較で q 文字マッチしたとすると,

r は少なくとも q 増加する. また, r は減少しない.

$r \leq |S|$ よりマッチの回数は $|S|$ 以下.

Z アルゴリズム

- $S = P\$T$ とする．($|P| \leq |T|$ とする)
  - $\$$ は P, T に現れない文字
- $Z_i(S)$ を計算する
  - $O(|P|+|T|)$ 時間
- $Z_i(S) = |P|$ ならば P は S の部分文字列
  i は必ず T の中になる (P は $\$$ を含まないから)
  $\Rightarrow P$ は T の部分文字列
- そのままだと $O(|P|+|S|)$ 領域だが，$O(|P|)$ にできる
  - $Z_i(S) \leq |P|$ なので，参照される Z_i は$O(|P|)$領域で格納可

索引検索

- Web検索のように，決まった文字列に対して何度も検索を行う場合は，索引検索の方が高速
- 単語索引
  - 決まった単語のみを検索できる
  - 索引のサイズが小さい
  - 転置ファイル
- 全文検索
  - 任意の部分文字列を検索できる
  - 索引が大きくなる
  - 接尾辞木，接尾辞配列

転置ファイル

- 文字列を単語(形態素)に分解
- 単語ごとに出現位置(出現文書)を列挙
- 出現回数も記憶

1 4 8 12 15 19

$T=$ いるか|いないか|いないか|いるか|いるいる|いるか

いるか: 3回 (1, 12, 19)
いないか: 2回 (4, 8)
いるいる: 1回 (15)

文字列検索の問題点

- 任意の文字列を検索したい
- 部分文字列の数 = ${}_nC_2 = O(n^2)$
- 全ての部分文字列を索引に格納

  ⇒索引サイズ: $O(n^2)$

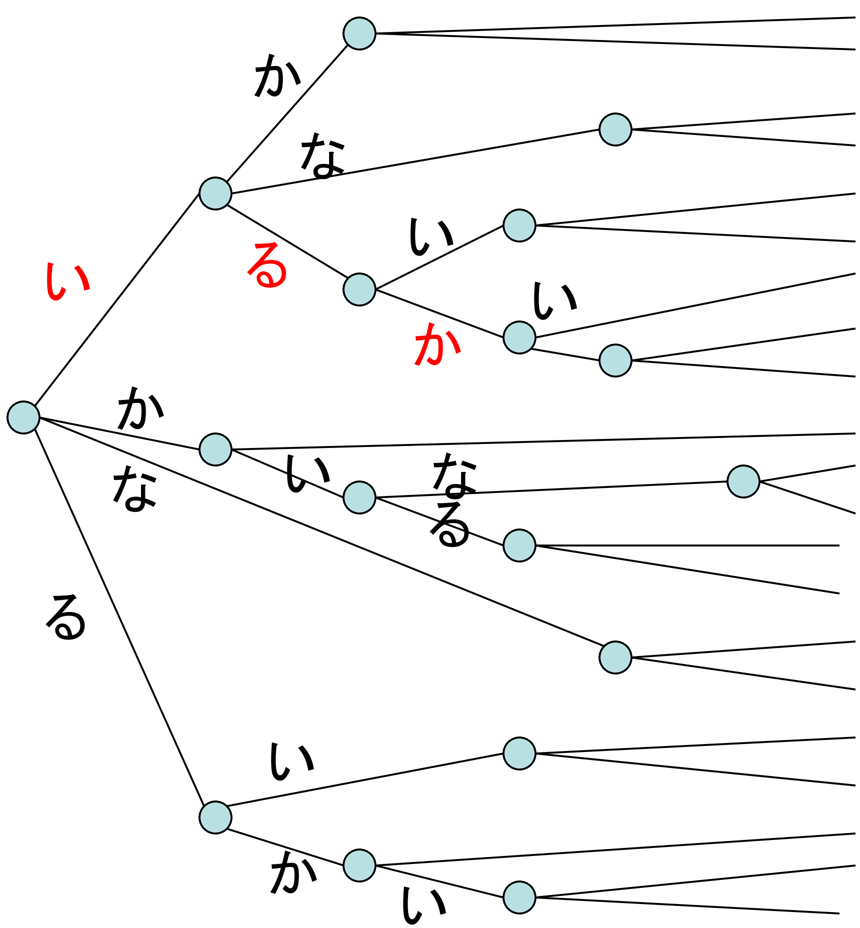
接尾辞 (suffix)

- 文字列 T の先頭の何文字を除いたもの (n 種類)
- T の任意の部分文字列は，ある接尾辞の接頭辞

1 いるかいないかいないかいるかいるいるいるか $= T$
2 るかいないかいないかいるかいるいるいるか
3 かいないかいないかいるかいるいるいるか
4 いないかいないかいるかいるいるいるか
5 ないかいないかいるかいるいるいるか
6 いかいないかいるかいるいるいるか
7 かいないかいるかいるいるいるか
8 いないかいるかいるいるいるか
9 ないかいるかいるいるいるか
10 いかいるかいるいるいるか
11 かいるかいるいるいるか
12 いるかいるいるいるか
13 るかいるいるいるか
14 かいるいるいるか
15 いるいるいるか
16 るいるいるか
17 いるいるか
18 るいるか
19 いるか
20 るか
21 か

接尾辞木 [Weiner 73]

- 全ての接尾辞を格納したcompacted trie



6 いかいないかいるかいるいるいるか
10 いかいるかいるいるいるか
4 いないかいないかいるかいるいるいるか
8 いないかいるかいるいるいるか
15 いるいるいるか
17 いるいるか
19 いるか
1 いるかいないかいないかいるかいるいるいるか
12 いるかいるいるいるか
21 か
3 かいないかいないかいるかいるいるいるか
7 かいないかいるかいるいるいるか
14 かいるいるいるか
11 かいるかいるいるいるか
5 ないかいないかいるかいるいるいるか
9 ないかいるかいるいるいるか
16 るいるいるか
18 るいるか
20 るか
2 るかいないかいないかいるかいるいるいるか
13 るかいるいるいるか

接尾辞配列 [Manber, Myers 93]

- 接尾辞のポインタを辞書順にソートした配列

1	いるかいないかいないかいるかいるいるいるか
2	るかいないかいないかいるかいるいるいるか
3	かいないかいないかいるかいるいるいるか
4	いないかいないかいるかいるいるいるか
5	ないかいないかいるかいるいるいるか
6	いかいないかいるかいるいるいるか
7	かいないかいるかいるいるいるか
8	いないかいるかいるいるいるか
9	ないかいるかいるいるいるか
10	いかいるかいるいるいるか
11	かいるかいるいるいるか
12	いるかいるいるいるか
13	るかいるいるいるか
14	かいるいるいるか
15	いるいるいるか
16	るいるいるか
17	いるいるか
18	るいるか
19	いるか
20	るか
21	か

SA	
6	いかいないかいるかいるいるいるか
10	いかいるかいるいるいるか
4	**いないか**いないかいるかいるいるいるか
8	**いないか**いるかいるいるいるか
15	**いるいる**いるか
17	**いるいる**か
19	**いるか**
1	**いるか**いないかいないかいるかいるいるいるか
12	**いるか**いるいるいるか
21	か
3	かいないかいないかいるかいるいるいるか
7	かいないかいるかいるいるいるか
14	かいるいるいるか
11	かいるかいるいるいるか
5	ないかいないかいるかいるいるいるか
9	ないかいるかいるいるいるか
16	るいるいるか
18	るいるか
20	るか
2	るかいないかいないかいるかいるいるいるか
13	るかいるいるいるか

接尾辞配列を用いた文字列検索

- 整数の配列での2分探索と同様に行う
- ただし，1回の比較は文字列同士の比較
  - 長さ m の文字列の比較なので $O(m)$ 時間
- $O(\log n)$ 回の文字列比較なので
  全体で $O(m \log n)$ 時間

接尾辞配列を用いた検索

- SA の上で二分探索

  $P =$ いるか

  3回 (1, 12, 19)

SA	
6	いかいないかいるかいるいるいるか
10	いかいるかいるいるいるか
4	いないかいないかいるかいるいるいるか
8	いないかいるかいるいるいるか
15	いるいるいるか
17	いるいるか
19	いるか
1	いるかいないかいないかいるかいるいるいるか
12	いるかいるいるいるか
21	か
3	かいないかいないかいるかいるいるいるか
7	かいないかいるかいるいるいるか
14	かいるいるいるか
11	かいるかいるいるいるか
5	ないかいないかいるかいるいるいるか
9	ないかいるかいるいるいるか
16	るいるいるか
18	るいるか
20	るか
2	るかいないかいないかいるかいるいるいるか
13	るかいるいるいるか

- P に対応する接尾辞配列の区間 $[s,t]$ が求まったら, P の出現位置を求めるのは容易
- P の出現回数 occ に比例した時間で列挙できる
- 検索全体の時間計算量は $\mathrm{O}(m \log n + occ)$

索引のサイズと検索時間

	サイズ	頻度問い合わせ時間
転置ファイル	$< n$ bytes	$O(m)$
接尾辞配列	$4n$ bytes + $\|T\|$	$O(m \log n)$
接尾辞木	$>10n$ bytes + $\|T\|$	$O(m)$

注: 転置ファイルは文書が単語に分かれている場合

圧縮接尾辞配列 (CSA)

- SA の代わりに $\Psi[i] = SA^{-1}[SA[i]+1]$ を格納
- サイズ: $O(n \log |A|)$ bits
- パタン P の検索: $O(|P| \log n)$ time

Ψ	i	SA	
0	**1**	**7**	\$
5	**2**	**1**	ababac\$
6	**3**	**3**	abac\$
7	**4**	**5**	ac\$
3	**5**	**2**	babac\$
4	**6**	**4**	bac\$
1	**7**	**6**	c\$

なぜ圧縮できるのか

- 接尾辞は辞書順に格納される
- 先頭の1文字を消しても辞書順は同じ

Ψ	*SA*	
12	6	いかいないかいるかいるいるいるか
14	10	いかいるかいるいるいるか
15	4	いないかいないかいるかいるいるいるか
16	8	いないかいるかいるいるいるか
17	15	いるいるいるか
18	17	いるいるか
19	19	いるか
20	1	いるかいないかいないかいるかいるいるいるか
21	12	いるかいるいるいるか
0	21	か
3	3	かいないかいないかいるかいるいるいるか
4	7	かいないかいるかいるいるいるか
5	14	かいるいるいるか
9	11	かいるかいるいるいるか
1	5	ないかいないかいるかいるいるいるか
2	9	ないかいるかいるいるいるか
6	16	るいるいるか
7	18	るいるか
10	20	るか
11	2	るかいないかいないかいるかいるいるいるか
13	13	るかいるいるいるか

CSA の性質

- $i < j$ のとき $T[SA[i]] \le T[SA[j]]$
- $i < j$ かつ $T[SA[i]] = T[SA[j]]$ のとき
  $\Psi[i] < \Psi[j]$

証明: $T[SA[i]] = T[SA[j]]$ のとき, この接尾辞の辞書順は2文字目以降で決まる.

$i < j$ より $T[SA[i]+1..n] < T[SA[j]+1..n]$

$SA[i'] = SA[i]+1$, $SA[j'] = SA[j]+1$ とすると, $i' < j'$

つまり $i' = SA^{-1}[SA[i]+1] = \Psi[i] < \Psi[j] = j'$

Ψ	i	SA	
0	**1**	**7**	\$
5	**2**	**1**	ababac\$
6	**3**	**3**	abac\$
7	**4**	**5**	ac\$
3	**5**	**2**	babac\$
4	**6**	**4**	bac\$
1	**7**	**6**	c\$

要素 $SA[i]$のアクセス方法

$n = 8$

$w = 3$

- i が $\log n$ の倍数のときに $SA[i]$ を格納
- $k = 0;\ \ w = \log n;$
- while ($i\ \ \%\ w\ != 0$)
  - $i = \Psi[i];\ \ k{+}{+};$
- return $SA_2[i\ /\ w]$ - k;

SA_2

0	8
1	3
2	4

Ψ	i	SA	
	0	**8**	
0	**1**	**7**	$
5	**2**	**1**	ababac$
6	**3**	**3**	abac$
7	**4**	**5**	ac$
3	**5**	**2**	babac$
4	**6**	**4**	bac$
1	**7**	**6**	c$

アクセス時間: 平均 $O(\log n)$ 時間

部分文字列の検索

二分探索時に実際のSAの値は必要ない

T	E	B	D	E	B	**D**	**D**	**A**	D	D	E	B	E	B	D	C
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Ψ	10	8	9	11	13	15	1	6	7	12	14	16	2	3	4	5
r	1	2	2	2	2	3	4	4	4	4	4	4	5	5	5	5
D	1	1	0	0	0	1	1	0	0	0	0	0	1	0	0	0
SA	8	14	5	2	12	16	7	15	6	9	3	10	13	4	1	11
	A	B	B	B	B	C	**D**	D	**D**	D	D	D	E	E	E	E
	D	D	D	D	E		**A**	C	**D**	D	E	E	B	B	B	B
		C	D	E					**A**	E	B	B	D	D	D	E
											D	E	C	D	E	

	1	2	3	4	5
C	A	B	C	D	E

後方検索

$C[\$]=[1,1]$
$C[\mathrm{a}]=[2,4]$
$C[\mathrm{b}]=[5,6]$
$C[\mathrm{c}]=[7,7]$

$P=P[1..p]$ の検索

$l \leftarrow 1;\ r \leftarrow n$

for ($k = p$; $k >=1$; k--)

$$l \leftarrow \mathop{\arg\min}_{j \in C[P[k]]} \Psi[j] \geq l$$

$$r \leftarrow \mathop{\arg\max}_{j \in C[P[k]]} \Psi[j] \leq r$$

$O(p \log n)$ time

Ψ	i	SA	
0	**1**	**7**	\$
5	**2**	**1**	ababac\$
6	**3**	**3**	abac\$
7	**4**	**5**	ac\$
3	**5**	**2**	babac\$
4	**6**	**4**	bac\$
1	**7**	**6**	c\$

$$l \leftarrow \arg\min_{j \in C[P[k]]} \Psi[j] \geq l$$

$$r \leftarrow \arg\max_{j \in C[P[k]]} \Psi[j] \leq r$$

- Ψの値で二分探索: $O(\log n)$ time
- P の検索に $O(|P| \log n)$ time

テキストの部分的な復元

T[9..13] = **DDEBE**を復元する場合

1. i=SA^{-1}[9]=10 を求める
2. 辞書順で i 番目の接尾辞の先頭の文字を求める
3. i=10からΨをたどる

	1	2	3	4	5
C	A	B	C	D	E

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	E	B	D	E	B	D	D	A	**D**	**D**	**E**	**B**	**E**	B	D	C
	A	B	B	B	**B**	C	D	D	D	**D**	D	**D**	**E**	E	E	**E**
Ψ	10	8	9	11	13	15	1	6	7	12	14	16	2	3	4	5
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA	8	14	5	2	12	16	7	15	6	9	3	10	13	4	1	11

圧縮接尾辞配列の機能

- *lookup*(i): $SA[i]$ を返す (O(log n) 時間)
- *inverse*(i): $SA^{-1}[i]$ を返す (O(log n) 時間)
- $\Psi[i]$: $SA^{-1}[SA[i]+1]$ を返す(O(1) 時間)
- *substring*(i,l): $T[SA[i]..SA[i]+l-1]$を返す
  – O(l) 時間
  – (i から$T[SA[i]$ は長さ n のベクトルの*rank*で求まる)