Li-Durbin Algorithm の解説

問題　文字列 X のなかに、問合せ W をミスマッチおよび挿入削除を z 個まで許して探す

$X =$ **AAATTTCCCGGGGGGG**

$W =$ **CCCTTTAAA**

$z = 1$

存在するか？

$X =$ **AAATTTCCCGGGGGGG**

$W =$ **TTTCGGGG**

$z =$ **1**

存在するか？

W= **CCCTTTAAA**

ミスマッチが起こるまで先頭から
BWT を使って探してゆく

```
AAATTTCCCGGGGGGG
      |||x
      CCCT

AAATTTCCCGGGGGGG
   ||x
   TTA

AAATTTCCCGGGGGGG
||
AA
```

最低2個のミスマッチは存在する
z=1 を満たす解は存在しないことを
高速に判断できる

元の配列を断片化し
各断片の位置をさがす戦略

CCCT TTA AA

ミスマッチ数の下限の1つである
2を高速に見積もれるが、
最小ミスマッチ数6を絞れない

```
AAATTTCCCGGGGGGG
      |||xxxxxx
      CCCTTTAAA

AAATTTCCCGGGGGGG
xxx|||xxx
CCCTTTAAA
```

W= **TTTCGGGG**

ミスマッチが起こるまで先頭から
BWT を使って探してゆく

```
AAATTTCCCGGGGGGG
    ||||x
    TTTCG
```

```
AAATTTCCCGGGGGGG
          |||
          GGG
```

最低1個のミスマッチは存在する
z=1 を満たす解は存在する可能性あり

惜しい！　最小ミスマッチ数は2

```
AAATTTCCCGGGGGGG
    ||||xx||
    TTTCGGGG
```

```
AAATTTCCCGGGGGGG
    ||||  ||||
    TTTC--GGGG
```

以下のように分解した断片を独立に
探すので、最小値を正確に求められる
わけでないが高速

TTTCG GGG

高速なので利用限界を理解して使う

復習：BWT を使って配列を検索する方法

S = **TATAATAATATAATA$**

C

$	A	C	G	T
0	1	10	10	10

	SA		BWT	Occ A	Occ C	Occ G	Occ T
0	15	$TATAATAATATAATA		1	0	0	0
1	14	A$TATAATAATATAAT		1	0	0	1
2	11	AATA$TATAATAATAT		1	0	0	2
3	3	AATAATATAATA$TAT		1	0	0	3
4	6	AATATAATA$TATAAT		1	0	0	4
5	12	ATA$TATAATAATATA		2	0	0	4
6	9	ATAATA$TATAATAAT		2	0	0	5
7	1	ATAATAATATAATA$T		2	0	0	6
8	4	ATAATATAATA$TATA		3	0	0	6
9	7	ATATAATA$TATAATA		4	0	0	6
10	13	TA$TATAATAATATAA		5	0	0	6
11	10	TAATA$TATAATAATA		6	0	0	6
12	2	TAATAATATAATA$TA		7	0	0	6
13	5	TAATATAATA$TATAA		8	0	0	6
14	8	TATAATA$TATAATAA		9	0	0	6
15	0	TATAATAATATAATA$					

Proposition:

$lb(xZ) = C(x) + Occ(x, lb(Z)-1)$

$ub(xZ) = C(x) + Occ(x, ub(Z)) - 1$

W = **TAA**

$lb(\{\}) = 0$

$lb(\mathtt{A}) = C(\mathtt{A})+Occ(\mathtt{A}, lb(\{\})-1) = 1$

$lb(\mathtt{AA}) = C(\mathtt{A})+Occ(\mathtt{A}, lb(\mathtt{A})-1) = 2$

$lb(\mathtt{TAA}) = C(\mathtt{T})+Occ(\mathtt{A}, lb(\mathtt{AA})-1) = 11$

$ub(\{\}) = 15$

$ub(\mathtt{A}) = C(\mathtt{A})+Occ(\mathtt{A}, ub(\{\}))-1 = 9$

$ub(\mathtt{AA}) = C(\mathtt{A})+Occ(\mathtt{A}, ub(\mathtt{A}))-1 = 5$

$ub(\mathtt{TAA}) = C(\mathtt{T})+Occ(\mathtt{T}, ub(\mathtt{AA}))-1 = 13$

実装（W の後方から計算）

```
k = 0;        // lb
l = |S|-1;    // ub
for(int i = |W|-1; 0 <= i; i--){
    k = C(W[i]) + Occ(W[i],k-1);
    l = C(W[i]) + Occ(W[i], l) -1;
    if( k > l ) leave;
}
```

TAA の存在が分かった後、**TAAA** や **TAAT** の存在を継続して調べにくい。
ゼロからやり直し。

BWT 内での計算順序
毎回ゼロからやり直し

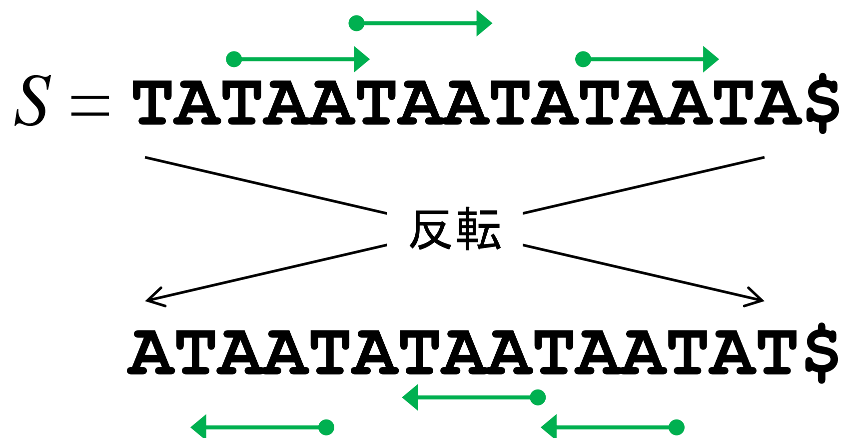
TA

TAA

TAAA

TAAT

Li-Durbin Algorithm　の着眼点



反転配列の Suffix array を構築

問合せと BWT の検索方向が一致

TAA の存在が分かった後、**TAAA** や
TAAT の存在を継続して調べやすい

Precalculation:

Calculate BWT string B for reference string X
Calculate array $C(\cdot)$ and $O(\cdot,\cdot)$ from B
Calculate BWT string B' for the reverse reference
Calculate array $O'(\cdot,\cdot)$ from B'

- 検索対象配列 X から BWT B, C, Occ を計算する。
- X を反転した配列から BWT B' , C', Occ' を計算する。ただし、C' と C は同一なので流用 。

Procedures:

InexactSearch(W, z)
CalculateD(W)
return InexRecur$(W, |W|-1, z, 1, |X|-1)$

- ミスマッチ数を最大 z 個許して、問合せ配列 W が X に出現する位置を suffix array の区間として枚挙する。
- 部分配列 $W[0,i]$ がX に最小 c 個のミスマッチで出現するとき、c の下限 $D(i)$ $(\leq c)$ を1つ計算。
- ミスマッチ許容数 z 個以下で、問合せ W の 0 から $|W|-1$番目の文字が、X の 1から $|X|-1$の範囲に出現する位置を suffix array の区間として枚挙。
- $D(i)$ の情報をうまく利用して効率化。
- $D(i)=0$ は自明。計算効率が上がらない。$D(i)$ はできるだけ大きく。

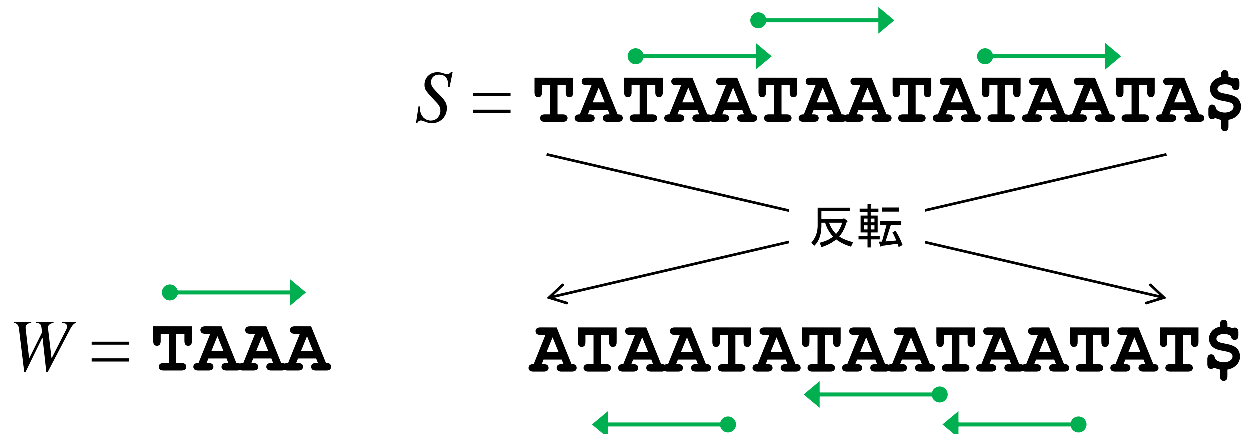
```
CalculateD(W)
  k = 0;
  l = |X|−1;
  z = 0;
  j = 0;

  for(int i = 0; i < |W|; i++){
    k = C(W[i]) + O'(W[i], k−1);
    l = C(W[i]) + O'(W[i], l) −1;

    if( k > l ){
      k = 0;
      l = |X|−1;
      z++;
      j = i + 1;
    }
   D(i) = z;
  }
```

- X の反転配列全域に、$[k, l]$ を初期化
- 問合せ W の部分配列 $W[j, i]$ が完全マッチで出現する suffix array の区間を計算
- $W[j, i]$ の i を1つ1つ増やし　$[k,l]$ の範囲を絞る
- $k > l$ の時は $W[i]$ でミスマッチが生じ、$W[j, i]$ は完全マッチしないことを意味
- ミスマッチ数 z を1 増やす。
- $j = i + 1$ として、$W[j, i]$ を、X 全域から検索

論文のままだと動かないので修正してあります



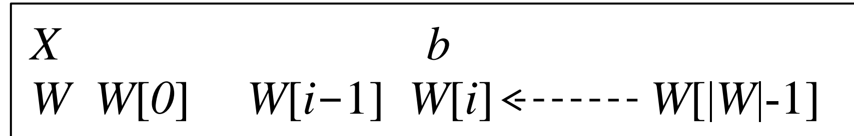
ミスマッチ許容数 z 個以下で $W[0, i]$ が出現する X 中の位置を SA の区間 $[k, l]$ の中に探す。

```
InexRecur (W, i, z, k, l)  {

  if ( z < 0  ||  (0 <= i && z < D[i]) )
    return φ;
  if ( i < 0)
    return { [k, l] };


  I = φ;
  I = I ∪ InexRecur( W, i−1, z−1, k, l)
  for each b in {A, C, T, G} do {
    int k1 = C[b]+O(b, k−1);
    int l1 = C[b]+O(b, l)−1;
    if ( k1 <= l1 ){
      I = I ∪ InexRecur(W, i, z−1, k1, l1);
      if ( b == W[i] )
        I = I ∪ InexRecur(W, i−1, z, k1, l1);
      else
        I = I ∪ InexRecur(W, i−1, z−1, k1, l1);
  }}
 return I; }
```

- i の初期値を $|W|-1$ で呼び、i を減らし W を後から前に処理し、0 になるまで再帰的に呼ぶ
- ミスマッチ数の下限 $D(i)$ が許容数 z を上回ると、条件を満たす位置は存在しない
- 探索を終了し、区間 $[k, l]$ を返す



- $W[i]$ は削除されたとみなす。∪ は和集合の意味
- 問合せ W の中身は事前に予測できないので b はすべて試す
- $W[i]$ は挿入された (b は空白) とみなす
- マッチしている場合、許容数 z はそのまま
- ミスマッチしている場合、許容数 z は1減らす

論文のままだと動かないので修正してあります

註） 重複を除く∪の実装が難しいので、重複を含んでいて構いません