接尾辞配列とその応用

nPk

Abstract–

Keywords: suffix array, in-place

1 記法

文字列は S とかき, 部分文字列は Rust の区間記法にしたがう. つまり, 表 1 の通りにする. 接尾辞配列 SA についても同じようにする.

S, T, ..	文字列
|S|	文字列の長さ
S[i]	文字
S[i..j], S[i..=j]	部分文字列
S[i..]	接尾辞

2 接尾辞配列

文字列 S の接尾辞 S[i..]は頭文字の添え字 i で指定できる. 接尾辞（に対応する添え字）を辞書順に並べたものを接尾辞配列という. 接尾字配列上で二分探索を行うと, 任意の文字列 T の全部の出現個所を $O(|T|\log|S|)$ で求めることができる. 検索文字列についてオンラインに処理できる点で接尾辞配列は Z アルゴリズムや KMP 法よりも優れている.

2.1 文字

Rust 言語において文字は `T: Ord` と抽象化できる. 接尾辞配列においてすべての接尾辞の頭文字がソートされていることを考えれば, 計算量は最悪 $\Omega(|S|\log|S|)$ であることが分かる. 実際に, $O(1)$ の作業メモリを使って, 文字列 `&[T]` の接尾辞配列を $O(|S|\log|S|)$ で計算するアルゴリズムが存在する.

文字列を座標圧縮することを考える. このとき, `T = usize` である. 後述するように, 文字列 `&[usize]` の接尾辞配列は $\Theta(|S|)$ で求めることができる. 座標圧縮の計算量は $\Theta(|S|)$ であるから, これによって最悪計算量が悪化することはない. [1] 本稿では, 文字は `usize` であるとする. [2]

[1] 基数ソートによる。

[2] より小さな数値型を用いても良いが、アルゴリズムが複雑になってしまう。本稿で紹介する実装では `Vec<T>` の最大容量が `isize::MAX` であることを利用した最適化を施しており、オリジナルと比べてアルゴリズムが単純になった。最上位ビットをフラグに使っているだけなので、2^{31} 文字以下ならば `u32` でも十分だが、コンパイル時の保証は得られない。

2.2 理論

2 つの接尾辞 S[i..]と S[i+1..]は長さが異なるので, 大小関係が決まる. そこで, 接尾辞およびその頭文字の型を次のように定義する.

定義 2.2.1: $\mathrm{S[i..]} \gneqq \mathrm{S[i+1..]}$ が成り立つとき, S[i]を L 型・S 型の接尾辞という. ただし, 末尾の番兵は S 型であるとする. L 型・S 型の接尾辞の頭文字も L 型・S 型とする.

定理 2.2.1: $\mathrm{S[i]} \gneqq \mathrm{S[i+1]}$ のとき, S[i]は L 型・S 型である. $\mathrm{S[i]} = \mathrm{S[i+1]}$ のとき, 両者の型は一致する.

証明　前者は自明. 後者については, $\mathrm{S[i+k]} \neq \mathrm{S[i+k+1]}$が成り立つ最小の正整数 k を考えればよい. □

接尾辞配列 SA は当然頭文字についてソートされる. 同じ頭文字 S[i]をもつ接尾辞たちが記録される部分をバケット S[i]と呼ぶことにする.

系 2.2.1: 各バケットは L 型接尾辞と S 型接尾辞で区切られている.

証明　頭文字を固定して考える. この頭文字をもつ最小の S 型接尾辞を S[i..]とかくと, $\mathrm{S[i]} < \mathrm{S[i+1]}$が成り立つ. 同様に最大の L 型接尾辞を S[j..]とかくと, $\mathrm{S[j]} > \mathrm{S[j+1]}$が成り立つ. したがって, $\mathrm{S[i+1]} > \mathrm{S[j+1]}$である. □

L 型接尾辞 S[i..]の性質より, $\mathrm{S[i..]} > \mathrm{S[i+1..]}$が成り立つ. したがって, バケット S[i]に書き込まれる L 型接尾辞の 2 文字目以降からなる接尾辞はバケットの左側にある. 同様に S 型接尾辞の 2 文字目以降からなる接尾辞は対応するバケットよりも右側にある. これらの観察より次の定理を得る.

定理 2.2.2: S 型接尾辞がソート済みならば, L 型接尾辞も $\Theta(|S|)$ でソートできる. 逆も成り立つ.

証明　L 型接尾辞をソートすることにする. 接尾辞配列を昇順に走査し, 初期化済みの要素を探す. これを SA[i]とかく. もし S[SA[i]−1..]が L 型接尾辞ならば対応するバケットに前から詰めて書き込む. 同じバケットに属する接尾辞はその 2 文字目以降の部分からなる接尾辞についてもソートされているから, 各バケットに書き込まれた L 型接尾辞はソートされて

いる. バケットソートと累積和でバケットの左端を求めることができるので, アルゴリズムは線形時間で動作する. 逆も同じように証明できる. □

定理 2.2.2 の証明では S 型接尾辞のうち, 1 だけ長い接尾辞が L 型であるもののみを利用している. これより, 次の事実を得る.

定義 2.2.2: S[i-1..]が L 型であるような S 型接尾辞 S[i..]をとくに LMS 型（leftmost S type）という. 文字 S[i]についても同様に定義する.

系 2.2.2: LMS 型接尾辞がソートされているとき, 接尾辞配列を $\Theta(|S|)$ で構築できる.

証明 LMS 型接尾辞がソートされているとする. 定理 2.2.2 より, 接尾辞配列 SA を昇順に走査して L 型接尾辞をソートできる. さらに, 接尾辞配列を降順に走査して S 型接尾辞をソートすることができる. □

補題 2.2.1: LMS 型接尾辞は高々 $\left\lfloor \frac{|S|}{2} \right\rfloor$ 個しかない.

補題 2.2.1 より考えるべき接尾辞の数を半分以下にできましたが, 問題サイズは半分になっていません. 接尾辞の長さが高々 $|S|$ だからです. LMS 型接尾辞を比較することを考えると, 2 つの LMS 型文字に挟まれてできる部分文字列ごとに比較すればよいことが分かります. これを利用して文字列を圧縮し, 問題のサイズを半分以下にすることが次の目標です.

定義 2.2.3: 部分文字列 S[i..=j]のうち, S[i]と S[j]のみが LMS 型であるものを LMS 型文字列という. とくに, 番兵も LMS 型部分文字列であるとする.

補題 2.2.2: LMS 型接尾辞と LMS 部分文字列の数は一致する.

LMS 部分文字列を順序を保ったまま新しい文字に置き換えるためには, これらをソートする必要があります. 逆に, LMS 部分文字列がソートされていれば

隣り合う 2 つの一致判定をとることで, 順序を保ったまま改名することができます.

定理 2.2.3: LMS 部分文字列を $\Theta(|S|)$ でソートできる.

証明　バケットソートにより, すべての LMS 文字を線形時間でソートできる. 系 2.2.2 の証明と同じようにして, 頭文字を除く最初の LMS 文字までソートできることが分かる. これはすべての LMS 部分文字列を含んでいる. □

系 2.2.3: 線形時間で接尾辞配列を構築できる.

証明　時間計算量は $T(n) \leq T(\frac{n}{2}) + \Theta(n) = \Theta(n)$ である. □

2.3 アルゴリズム

本節では線形時間接尾辞配列構築アルゴリズムの in-place な実装を紹介する. 作業メモリを節約するために, バケットソートや区間幅の管理に接尾辞配列 SA 用のメモリを利用する. 2 章 1 節 で述べたように, 文字の最上位ビットを活用して接尾辞の添え字とカウンターを区別する. また, 文字の型の区別にも利用する.

2.3.1 文字の改名

文字列が座標圧縮されているので, 接尾辞配列を使って各文字の登場回数を数えることができる. [3] 累積和をとると, バケットの右端を得ることができる. 系 2.2.1 より, L 型文字をバケットの左端, S 型文字をその右端の添え字に改名してもよい.

アルゴリズム 2.3.1.1:

```rust
fn rename(text: &mut [usize], sa: &mut [usize]) {
    // 文字の登場回数を数える
    sa.fill(0);
    text.iter().for_each(|s| sa[*s] += 1);
    // 累積和をとり, バケットの右端を求める.
    (1..sa.len()).for_each(|i| sa[i] += sa[i-1]);
```

[3] 実際には $S[i] < |S|$ で十分。

```
    // 改名する.
    {
      // 番兵はS型. その1つ前はL型.
      let mut is_s_type = false;
      // 番兵は1つしかないので, 改名しても`0`のまま.
      // 直前の要素を型の判別に使うので, 更新を遅延する.
      let mut name = [0, 0];
      for i in (0..text.len()).rev().skip(1) {
        if text[i] != text[i+1] {
          is_s_type = text[i] < text[i+1]
        }
        // 遅延を反映する
        if let Some(s) = text.get_mut(i+2) {
          *s = name[i%2]
        }
        // 反映を遅延する
        name[i%2] = if is_s_type {
          sa[text[i]]-1
        } else {
          // L型は最上位ビットを立てておく.
          sa[text[i]-1] | 1usize.rotate_right(1)
        };
      }
      if text.len() >= 2 {
        text[..2].copy_from_slice(&name);
      }
    }
}
/// 文字の型を判定する.
fn is_s_type(s: usize) -> bool {
  s & 1usize.rotate_right(1) == 0
}
/// L型文字ならバケットの左端, S型文字ならバケットの右端の添え字を返す.
```

```rust
fn str_to_ptr(s: usize) -> usize {
  s & (!0 >> 1)
}
```

文字の最上位ビットをフラグに使うことで, 文字の型をいつでも定数時間で求めることができるようになった. 定理 2.2.1 から文字の型を判別するには S を逆順に走査する必要がある. したがって, 系 2.2.2 接尾辞配列を操作する際にはこの手法は利用できない. [4] また自由な方向に走査できるので, メモリ IO の回数を削減できるかもしれない.

2.3.2 LMS 文字のソート

LMS 部分文字列をソートするために, LMS 文字をソートする. LMS 文字をバケットの正しい位置に置く際に注意が必要である. LMS 文字はバケットの右端の情報をもつが, 愚直に書き込むと $O(|S|^2)$ かかってしまう.

そこで, バケットの右端には次に書き込むべき場所への距離を書き込むことにする. これはすでに書き込んだ文字の個数に 1 を加えたものである. バケット内に LMS 文字が 1 つしかない場合, カウンターは不要なので単に書き込んでしまう. これと区別するために 1 を加えた.

LMS 文字についてソートされていればよいので, バケット内での順序は自由にしてよい. 系 2.2.2 では文字でなく接尾辞の順序を保つので, 全体をシフトする必要がある.

アルゴリズム 2.3.2.1:

```rust
fn sort_lms_char(text: &[usize], sa: &mut [usize]) {
  // LMS文字を数える.
  sa.fill(0);
  text.windows(2).for_each(|s| {
    if !is_s_type(s[0]) && is_s_type(s[1]) {
      sa[str_to_ptr(s[1])] += 1
    }
  });

  // カウンターの最上位ビットを立てて区別する. 実際はグローバルな定数.
```

[4] 別の方法はある。

```
  const COUNTER_FLAG: usize = 1.rotate_right(1);
  const COUNT_ZERO: usize = COUNTER_FLAG;
  const COUNT_ONE: usize = COUNT_ZERO + 1;
  text.windows(2).enumerate().for_each(|(i, s)| {
    if !is_s_type(s[0]) && is_s_type(s[1]) {
      let ptr = str_to_ptr(s[1]);

      if sa[ptr] == COUNT_ONE {
        // 唯一の要素なので, カウンターは不要
        sa[ptr] = i+1;
      } else if sa[ptr-1] == COUNT_ZERO {
        // バケット内で最初の文字
        sa[ptr-1] = i+1;
        // 引き算だとフラグが消えてしまう
        sa[ptr] = COUNT_ZERO+2;
      } else {
        // バケット内で 2 文字目以降
        let diff = sa[ptr] - COUNT_ZERO;
        if sa.get(ptr.wrapping_sub(diff)).is_some_and(|sa| *sa == COUNT_ZERO) {
          // 次に書き込む先はバケット内か, L型用のバケット
          sa[ptr-diff] = i+1;
          sa[ptr] += 1;
        } else {
          // 隣のS型用のバケットにはみ出してしまう.
          // バケット内で最後の 1 文字なので, カウンターを上書きしてよい.
          sa[ptr] = i+1
        }
      }
    }
  });
  // カウンターを除去して, 右に 1 だけシフトする.
  {
```

```
    let mut i = sa.len();
    while i > 0 {
      i -= 1;
      if sa[i] > COUNT_ONE {
        // カウンターを最後のLMS文字で上書きする.
        let diff = sa[i] - COUNT_ZERO - 1;
        sa[i] = sa[i - diff];
        sa[i - diff] = COUNT_ZERO;
        // LMS文字がある部分はスキップする.
        i -= diff;
      }
    }
}
```

2.3.3 誘導ソート

系 2.2.2 の証明で用いたアルゴリズムを誘導ソートという. 誘導ソートはこのアルゴリズムの肝であり, ソート済みの LMS 文字から LMS 部分文字列の順序を導き, ソートされた LMS 型接尾辞から接尾辞配列を誘導する.

アルゴリズム 2.3.3.1 (Induced-Sort):

```rust
fn induced_sort(text: &[usize], sa: &mut [usize]) {
  // LMS型からL型を誘導する
  sort_l_types(text: &[usize], sa: &mut [usize]);
  // LMS文字を除去する. 番兵は誘導できないので, 残しておく.
  sa.iter_mut().skip(1).for_each(|i| {
    // L型以外の文字は除去してよい.
    if text.get(*i).is_some_and(|s| is_s_type(*s)) {
      *i = COUNT_ZERO
    }
  });
  // L型からS型を誘導する
  sort_s_types(text: &[usize], sa: &mut [usize]);
```

```
13 }
14
15 // S型の誘導ソートもほとんど同じだが, SAを降順に走査する. カウンターは残らないので, あとから削除する必要はない.
16 fn sort_l_types(text: &[usize], sa: &mut [usize]) {
17   // LMS文字をソートするときとほとんど同じ. 違いはSではなくSAを昇順に走査すること.
18
19   // L型文字の出現数を数える
20   text.iter().for_each(|s| {
21     if !is_s_type(*s) { sa[str_to_ptr[*s]] += 1 }
22   });
23   // バケットに昇順に書き込む.
24   // バケットの中身を1だけシフトしたときには添え字を進めないので, forループは使えない.
25   let mut i = 0;
26   while i < sa.len() {
27     // 初期化済みで, 1つ長い接尾辞が存在するもの
28     if sa[i] > 0 && sa[i] & COUNTER_FLAG == 0 {
29       let s = text[sa[i]-1];
30       let ptr = str_to_ptr(s);
31       // シフトしても影響がなかった場合, 同じ操作はしない.
32       if !is_s_type(s) && sa[ptr] & COUNTER_FLAG != 0 {
33         if sa[ptr] == COUNT_ONE {
34           // バケット内で唯一のL型文字
35           sa[ptr] = sa[i]-1
36         } else if sa[ptr+1] == COUNT_ZERO {
37           // バケット内で最初のL型文字
38           sa[ptr+1] = sa[i]-1;
39           sa[ptr] = COUNT_ZERO + 2;
40         } else {
41           let diff = sa[ptr] - COUNT_ZERO;
42           if sa.get(ptr + diff).is_some_and(|sa| *sa == COUNT_ZERO) {
43             // 書き込まれていない場合
```

```
44                sa[ptr+diff] = sa[i]-1;
45                sa[ptr] += 1;
46              } else {
47                // 書き込まれているなら, バケット内で最後のL型文字
48                sa[ptr] = sa[i]-1;
49                // 順番を保つ
50                sa[ptr..ptr+diff].rotate_left(1);
51                // 今いる場所がシフトした場合, 添え字をインクリメントしてはけない
52                continue;
53              }
54            }
55          }
56          i += 1;
57        }
58    }
59    // カウンターを除去して, バケットを左にシフト
60    let mut i = 0;
61    while i < sa.len() {
62      if sa[i] > COUNT_ZERO {
63        // カウンターを初期化して左にシフト
64        let diff = sa[i] - COUNT_ZERO;
65        sa[i] = COUNT_ZERO;
66        sa[i..i + diff].rotate_left(1);
67        // L型文字が書き込まれている部分はスキップ
68        i += diff
69      } else {
70        i += 1
71      }
72    }
73  }
```

2.3.4 LMS 部分文字列のソート

LMS 文字はソート済みなので, LMS 部分文字列をソートできる. 部分問題を構築するメモリを確保するために, LMS 型の文字を SA の末尾に集める. LMS 文字列を改名したら, S での登場順に並び替え, 部分問題を再帰的に解く.

LMS 文字は高々 $\left\lfloor \frac{|S|}{2} \right\rfloor$ 個しかないので, 改名後の文字は SA[SA[i]/2]に書き込めばよい. 部分問題は SA の先頭に詰めて書き込まれ, その接尾辞配列は末尾に書き込まれる.

アルゴリズム 2.3.4.1:

```rust
/// LMS部分文字列を改名して, 新しい text と sa を返す. これを再帰的に解く.
fn sort_lms_substrings(text: &[usize], sa: &mut [usize]) -> (&mut [usize], &mut [usize]) {
  induced_sort(text, &mut sa);
  // LMS文字を末尾に集める.
  let n_lms = {
    let mut n_lms = 0;
    for i in (0..sa.len()).rev() {
      // text[sa[i]]はLMS型
      if sa[i] > 0 && is_s_type(text[sa[i]]) && !is_s_type(text[sa[i]-1]) {
        n_lms += 1;
        sa[sa.len()-n_lms] = sa[i]
      }
    }
    // 不要なノードは初期化
    let r = sa.len()-n_lms;
    sa[..r].fill(COUNT_ZERO);

    n_lms
  };
  // LMS部分文字列を改名する.
  let kind_lms = {
    let mut kind_lms = 0;
    let i = sa.len()-n_lms;
    // これは番兵
```

```
25      sa[sa[i]/2] = 0 // text[sa[i]]
26      for i in i+1..sa.len() {
27        // LMS部分文字列の長さを求める
28        let nl = text[sa[i-1]..].windows(2).take_while(|s| {
29                    // 次のLMS型まで進む
30                    is_s_type(s[0]) || !is_s_type(s[1])
31                  }).count();
32        let nr = text[sa[i]..].windows(2).take_while(|s| {
33                    // 次のLMS型まで進む
34                    is_s_type(s[0]) || !is_s_type(s[1])
35                  }).count();
36        // 一致しないなら次に大きな文字を割り当てる.
37        if text[sa[i-1]..sa[i-1]+nl] != text[sa[i]..sa[i]+nr] {
38          kind_lms += 1;
39        }
40        sa[sa[i]/2] = kind_lms
41      }
42
43      kind_lms
44    };
45    // 改名したLMS部分文字列を前に集める. これはtextに現れた順になっている.
46    {
47      let mut n = 0;
48      for i in 0..sa.len()-n_lms {
49        if sa[i] & COUNTER_FLAG == 0 {
50          sa[n] = std::mem::replace(&mut sa[i], COUNT_ZERO);
51          n += 1;
52        }
53      }
54    }
55
56    let (pre, sa) = sa.split_at_mut(sa.len()-n_lms);
57    let (text, _) = sa.split_at_mut(n_lms);
```

```
58    (text, sa)
59  }
```

2.3.5 LMS 接尾辞のソート

LMS 部分文字列の接尾辞配列が得られたので, LMS 接尾辞がソートできた. ここで, 部分問題が SA の前半に, その接尾辞配列が SA の後半に詰めて書き込まれているとする. 接尾辞配列を正しいバケットに格納するために, SA の後半にソートされた LMS 型接尾辞を詰める. その語, 昇順にバケットに書き込む.

アルゴリズム 2.3.5.1:

```rust
fn sort_lms_suffixes(text: &[usize], sa: &mut [usize])
{
  // 部分問題をLMS型接尾辞で登場順に上書きする
  let mut n = 0;
  text.windows(2).enumerate().for_each(|(i, s)| {
    if !is_s_type(s[0]) && is_s_type(s[1]) {
      sa[n] = i+1;
      n += 1;
    }
  });
  // 部分問題の解から, LMS型接尾辞をソートする.
  {
    let l = sa.len()-n;
    for i in l..sa.len() {
      sa[i] = sa[sa[i]];
    }
    sa[..n].fill(COUNT_ZERO);
  }
  // LMS型接尾辞をバケットに書き込む
  {
    n = 0;
    // 番兵は最小の接尾辞
    let mut s = 0;
```

```
    sa[0] = sa[l]
    for i in l+1..sa.len() {
      // バケット内でのオフセット
      if text[sa[i]] == s { n += 1 } else {
        // 降順で書き込んだので反転する
        sa[str_to_ptr(s)-n..=str_to_ptr(s)].reverse();
        n = 0;
      };
      s = text[sa[i]];
      // 同じ場所に書き込むことがあるので, 初期化してから上書きする.
      sa[str_to_ptr(s)-n] = std::mem::replace(&mut sa[i], COUNT_ZERO);
    }
    sa[str_to_ptr(s)-n..=str_to_ptr(s)].reverse();
  }
}
```

2.3.6 まとめ

アルゴリズムの全体は下記の通り. 参照は長さをもつスマートポインターなので, 実は $O(\log|S|)$ の作業メモリを使っている. 作業メモリを $O(1)$ に抑えるためには生ポインターを使う必要がある. これは unsafe rust の領分なので, 本稿では解説しない.

アルゴリズム 2.3.6.1 (Create-Suffix-Array):

```rust
fn suffix_array(text: &mut [usize], sa: &mut [usize]) {
  assert_eq!(text.len(), sa.len());
  todo!("base case");

  rename(&mut text, &mut sa);
  sort_lms_char(&text, &mut sa);
  {
    let (text, sa) = sort_lms_substrings(&text, &mut sa);
    suffix_array(&mut text, &mut sa);
```

```
10    }
11    sort_lms_suffixes(&text, &mut sa);
12    induced_sort(&text, &mut sa);
13  }
```

2.4 非再帰アルゴリズム

紹介したアルゴリズムの再帰木は鎖型になっている. 行きがけと帰りがけでループを 2 つ用意すると非再帰で書けるはずである.

借用規則を守るために `slice.split_at_mut()` を多用することになる. この可変参照を管理するためにスタックを使うかも？

2.5 制約の緩和

本稿では座標圧縮を仮定したが, 文字列の種類が $O(|S|)$ であれば, 線形時間で接尾辞配列をもとめるアルゴリズムが存在する. ASCII 文字や Unicode 文字など, コンピューターで利用できる文字の種類は固定なので, 接尾辞配列を線形時間で計算できる.

2.6 `String` 型

バイト列とみなして接尾辞配列を計算し, 入力を復元する. UTF-8 の規格から, エントリーポイントかどうかは `u8.leading_ones()` で判別できる.

3 LCP 配列

文字列 S の 2 つの部分文字列の最長共通接頭辞（Longest Common Prefix, LCP）を線形時間で構築するアルゴリズムを紹介する. 部分文字列は接尾辞の接頭辞なので, 対応する 2 つの接頭辞の LCP を計算出来ればよい.[5] 接尾辞配列がソート済みであることから次の事実を得る.

補題 3.1: 接尾辞 S[SA[i]±1..]の少なくとも一方は接尾辞 S[SA[i]..]との LCP が最大の接尾辞である.

[5] 部分文字列 S[i..i+ni]と S[j..j+nj]の LCP は `lcp(S[i..], S[j..]).min(ni).min(nj)` とかける。

系 3.1: 2つの接尾辞 S[SA[i]..]と S[SA[j]..]を考える. ただし, SA[i] < SA[j]とする. 次の関係が成り立つ.

$$\mathrm{lcp}(\mathrm{S[SA[i]..]}, \mathrm{S[SA[j]..]}) = \min_{\mathrm{SA[i]} \le k < \mathrm{SA[j]}} \mathrm{lcp}(\mathrm{SA[k..]}, \mathrm{SA[k+1..]}) \quad (1)$$

系 3.1 より接尾辞配列で隣り合う2つの接尾辞の LCP を計算し, それから RMQ を構築すればよい. RMQ の実装は省略するが, 構築 $O(N)$でクエリ $O(1)$ のものが存在する. また, 次の事実が成り立つ.

補題 3.2: 2つの接尾辞 S[SA[i]..]と S[SA[i+1]..]の LCP の長さを L とする. S[SA[i]+1..]と S[SA[i+1]+1..]の LCP の長さは $L-1$ 以上である.

この性質を使って LCP を高速に計算するためには接尾辞配列ではなく文字列 S を正順で走査するとよい.

アルゴリズム 3.1:

```rust
/// 返り値を lcp とすると, lcp[i] は接尾辞 text[i..] とその次に小さな接尾辞のLCPを表す.
fn lcp_array(text: &[usize], sa: &[usize]) -> Vec<usize> {
  // sa の逆関数で初期化. text を正順に走査しつつ, sa での順序を求めたい.
  let mut lcp = Vec::with_capacity(sa.len());
  {
    let lcp = lcp.spare_capacity_mut();
    (0..sa.len()).for_each(|i| { lcp[sa[i]].write(i); });
  }
  unsafe { lcp.set_len(sa.len()) };

  let mut l = 0;
  for i in 0..sa.len() {
    if lcp[i] > 0 {
      // 1つ小さな接尾辞.
      let j = sa[lcp[i]-1];
      // 少なくとも一方は Some(*)
```

```
17          while text.get(i+l) == text.get(j+l) { l += 1 };
18          // lcp[i] はもう不要なので, LCPの長さを書き込む. メモリの節
            約.
19          lcp[i] = l;
20          l = l.saturating_sub(1);
21      }
22    }
23
24    lcp
25  }
```

定理 3.1: LCP 配列を線形時間で構築できる.

証明　接尾辞配列は線形時間でっ構築できる. アルゴリズム 3.1 より, 接尾辞配列があれば線形時間で LCP 配列を計算できる. なぜなら, l は高々 $|S|$ であり, 高々 $|S|$ 回デクリメントされるので, while ループが高々 $2|S|$ 回実行されるから. □

LCP 配列を接尾辞の昇順に並べ替えるためには, SA[i]と LCP[SA[i]]をスワップすればよい. このとき, SA は接尾辞の昇順の LCP 配列となり, LCP は接尾辞配列の逆関数になっている.

4 接尾辞木

接尾辞木とは文字列 S のすべての接尾辞からなる Trie 木である. これは $O(|S|^2)$ のメモリを使用するが, パトリシア木のように $\Theta(|S|)$ 圧縮できる. 接尾辞配列と LCP 配列があるとき, 接尾辞木での深さ優先探索をシミュレートできることが知られている. この手法はメモリアクセスが連続的であり, 辺を明示的に管理しなくてよい点で優れている. 本稿では接尾辞木は圧縮されているものとする.

補題 4.1: （圧縮された）接尾辞木のノード数は高々 $2|S| - 1$ である.

証明　接尾辞を昇順に追加することを考える. 最初を除いて, 接尾辞を 1 つ追加する度にノードが 1 つか 2 つ増える. □

補題 4.1 は根を含まない. 接尾辞配列を頭文字で二分探索すれば部分木の根を得る.

定理 4.1: 深さ優先探索の計算量は $\Theta(|S|)$ である.

DFS のアルゴリズム. 帰りがけは簡単だが, 行きがけは難しそう

5 応用

5.1 文字列の検索

5.2 文字列の登場回数

5.3 ２つの文字列の最長共通部分列

ユニークな区切り文字 # と番兵 0 について, 新しい文字列 S + # + T + 0 を構築し, 接尾辞配列と LCP 配列を計算する. 接尾辞の長さと区切り文字がユニークであることから, S の接尾辞と T の接尾辞が区別できる. 接尾辞配列で隣接する S の接尾辞と T の接尾辞の LCP が最大のものが解である.

5.4 部分文字列の種類数

6 参考文献

6.1 接尾辞配列の線形時間構築

- LI, Zhize; LI, Jian; HUO, Hongwei. Optimal in-place suffix sorting. Information and Computation, 2022, 285: 104818.
- NONG, Ge; ZHANG, Sen; CHAN, Wai Hong. Two efficient algorithms for linear time suffix array construction. IEEE transactions on computers, 2010, 60.10: 1471-1484.

6.2 LCP 配列の線形時間構築と接尾辞木のシミュレーション

- KASAI, Toru, et al. Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Annual Symposium on Combinatorial Pattern Matching. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. p. 181-192.