第 3 章

すばやい文字列の探しかた

72 回生 Hinata

3.1　はじめに

皆さんこんにちは。72 回生の Hinata です。ちょっと前まで中学生だったのに、いつのまにか高校 2 年生になってしまいました。入学してから 4 年が経ちますが、自分が成長している気が全くせず厳しい気持ちでいます。今年は JOI(情報オリンピック) も落ちてしまいましたし。

さて、今回のタイトルは「すばやい文字列の探しかた」です。どっかで聞いたことあるようなタイトルですね。あまり上手なパロディではないですが、そこはご容赦ください。ちなみに冴えカノは見たことも読んだこともありません。読みたいとは思ってるんですけどお金がないんですよ。ラノベに限らず読みたい本が多すぎてつらいので、通りすがりの石油王がいらっしゃいましたら寄付をお願いします。

Web サイトを見ているとき、あるいはテキストエディタで文章 (あるいはコード) を書いているとき、ブラウザやテキストエディタの検索機能を使えば、長い文章やサイトでも一瞬で探し当ててくれます。では、この処理はどうやって実現されているのでしょうか。大多数の人は詳しく知らないでしょう。私もつい最近まで知りませんでした。

そこで、この記事では、文章から文字列 (文字の列、すなわち単語や文章など) を高速に検索するアルゴリズムをいくつか紹介・実装してみます。使う言語は Ruby です。

3.2　約束

この記事において、文字列 S の長さは $|S|$ と表記します。

また、文字列 S の部分列のうち、i 文字目から|S|文字目までを取り出した文字列を接尾辞と呼ぶことにします。例えば"ABCDEF"の接尾辞には"CDEF"や"BCDEF"などがあります。反対に、1 文字目から i 文字目までを取り出した文字列を接頭辞と呼ぶことにします。例えば"ABCDEF"の接頭辞には"ABC"や"ABCDE"などがあります。

3.3　とりあえず力技で

文字列を検索するアルゴリズムのうち、もっとも単純なのが総当たり (ブルートフォース) です。

長い文字列 S と検索したい文字列 W があるとします。S と W を 1 文字ずつずらして比較し、全て同じであれば成功です。途中で異なる文字があった場合は、S の開始位置を 1 つずらして同じことをします。

一回の比較に $O(|W|)$ かかり、その処理を $|S|$ 回繰り返すため、全体の計算量は $O(|S||W|)$ です。

実際に、S に含まれる W の開始位置を配列で返す関数を実装してみます。

List 3.1: ブルートフォース

```ruby
def brute_force(s, w)
  match_index = []

  0.upto(s.size - w.size) do |i|
    match = true
    0.upto(w.size - 1) do |j|
      if s[i + j] != w[j]
        match = false
        break
      end
```

```
    end
    match_index.push(i) if match
  end

  match_index
end
```

3.4 KMP 法

総当たりは無駄が多いです。

例えば、S が"BABABABABDC"で W が"ABABD"の場合を考えます。

	B	A	B	A	B	A	B	A	B	D	C
1	A										
2		A	B	A	B	D					
3			A								
4				A	B	A	B	D			
5					A						
6						A	B	A	B	D	
7							A				
8								A	B	A	
9									A		
10										A	
11											A

図 3.1: ブルートフォース

後戻りしてすでに探索した部分から比較を始めている部分がありますね。この比較をどうにか省いて高速化できないでしょうか。

ここで、1 から $|W|$ までの全ての i に対して、「i 文字目で終わり、W の接頭辞のひとつと一致する W の部分文字列のうち、最長のもの」を考えます。ただし、文字列そのものは含みません。

例えば、i が 4 だとします。3 文字目から始まり 4 文字目で終わる部分文字列"AB"は先頭の 2 文字"AB"と同じですね。今回はこれ以外に条件を満たす文字列がないため、最長のものは"AB"となります。

全ての i について最長の文字列の長さを計算すると、次のようになります。

表 3.1: 「ずらし表」

i	1	2	3	4	5
文字列長	0	0	1	2	0

では、実際に探索するとどうなるか考えてみましょう。ずらし表において i に対応する文字列長を t[i] と定義します。

例えば、"ABAB"まで一致して"D"が一致しなかった場合を考えます。t[4] は 2 ですが、これは"ABAB"の最後 2 文字が先頭 2 文字と等しいということなので、次の比較は途中で終わり 2 回先の比較でようやく"AB"と一致するわけです。

	B	A	B	A	B	A	B	A	B	D	C
1	A										
2		A	B	A	B	D					
3				A	B	A	B	D			
4						A	B	A	B	D	
5											A

図 3.2: KMP 法

灰色の部分は実際には実行されませんが、分かりやすさのために書いています。
実際のコードはこうなります。

List 3.2: KMP 法 (検索部)

```
def kmp(s, w)
  match_index = []

  # 「ずらし表」を作成する
  t = create_table(w)

  i = 0 # Sの比較開始位置
  j = 0 # Wの比較位置

  while i + j < s.size
    if s[i + j] == w[j]
      # 一致した場合
      j += 1

      if j == w.size
        # 検索成功
        match_index.push(i)

        # i = i + j - t[j - 1]
        # j = t[j - 1]
        i, j = i + j - t[j - 1], t[j - 1]
      end
    elsif j == 0
      # 1文字目で比較失敗
      i += 1
    else
      i, j = i + j - t[j - 1], t[j - 1]
    end
  end

  match_index
end
```

3.5 接尾辞配列

文字列 S の接尾辞を全て列挙することを考えます。開始位置 i が 0 から $|S| - 1$ まで動き、1 つの i に対して接尾辞 1 つが対応するので、S の接尾辞は $|S|$ 種類存在します。

さて、$|S|$ 種類の接尾辞を辞書順に並べた配列があったとします。これが接尾辞配列です。例えば、S が"abracadabra"の場合は次のようになります。

"bra"が出現する位置を全て探すということは、つまり"bra"から始まる接尾辞を探索することです。配列はソートされているので、二分探索を利用することで効率的に探すことができます。この場合、"bra"が出てくるのは 2 文字目と 9 文字目からです。

素朴な実装は次のようになります。

List 3.3: キャプション

```
class SuffixArray
  attr_reader :sa

  def initialize(str)
    @str = str
    @sa = Array.new(str.size) { |i| i }
    @sa.sort! { |i, j| str[i..-1] <=> str[j..-1] }
  end
end
```

表 3.2: 接尾辞配列

開始位置	接尾辞
11	a
8	abra
1	abracadabra
4	acadabra
6	adabra
9	bra
2	bracadabra
5	cadabra
7	dabra
10	ra
3	racadabra

```
p SuffixArray.new("abracadabra").sa # [10, 7, 0, 3, 5, 8, 1, 4, 6, 9, 2]
```

初めに [0,1,2,3,.....] という配列を生成して、それをソートしています。

さて、このコードの計算量はどうなっているでしょうか。ソートの計算量自体は $O(|S|log|S|)$ ですが、1 回の比較は $O(|S|)$ なので、接尾辞配列を構築する処理の計算量は $O(|S|^2log|S|)$ となります。これは、さすがに使い物になりません。実際、16.75 万文字の文章でこのコードを実行したところ、私のパソコンでは 5.3 秒かかりました。多少高速化はできますが、計算量が大きすぎて焼け石に水です。

しかし、実際には SA-IS 法という高速な構築法があり、 $O(|S|)$ で計算することができます。しかし、筆者のアルゴリズムを理解し実装する能力はとても低く、実装することも理解することもできませんでした。

Multi-key quicksort

というわけで、今回は Multi-key quicksort というアルゴリズムを実装してみます。これは文字列の比較に特化したクイックソートです。なぜこのアルゴリズムかというと、私は比較的再帰アルゴリズムが得意だからです。あと 9 時間で文化祭始まっちゃうのに、苦手な実装をしている余裕はないんですよ。(ヤバイ)

"abracadabra"の接尾辞は以下です。

```
abracadabra
bracadabra
racadabra
acadabra
cadabra
adabra
dabra
abra
bra
ra
a
```

ランダムに基準の文字列 (今回は adabra) を取ります。そして、1 文字目が a より前か、同じか、後ろかで分別します。

```
-----------
abracadabra
acadabra
adabra
abra
a
-----------
bracadabra
racadabra
cadabra
dabra
bra
ra
```

それぞれの区域をソートすれば、全体としてもソートできたことになりますね。

まず、a から始まる文字列をソートします。この場合、2 文字目で比較します。abracadabra を選択すると

```
[a]
-----------
[a]bracadabra
[a]bra
-----------
[a]cadabra
[a]dabra
```

この後も再帰的にソートしていきます。

さて、初めの文字が a よりも後ろだった文字列のソートはどうなるでしょうか。まだ 1 文字目は揃っていないので、もう一度 1 文字目で分別することになります。基準に bracadabra を選択すると

```
-----------
bracadabra
bra
-----------
racadabra
cadabra
dabra
ra
```

これも、それぞれの区域をソートすることで全体をソートできます。

文字列の比較は当然 $O(1)$ なので、全体の計算量は $O(|S|log|S|)$ です。多分。

コード書いた方が分かりやすいと思うので、サンプルコードを載せます。なお、このコードは速度を考慮していないため、さっきの力技ソートより遅いです。

List 3.4: キャプション

```ruby
class String
  def at_banpei(index)
    if self[index].nil?
      "\0"
    else
      self[index]
    end
  end
end

class SuffixArray
  attr_reader :sa
  def initialize(str)
    @str = str
    @sa = Array.new(str.size) { |i| i }
    @sa = qsort_sa(0, @sa)
  end

  # nは何文字目を比較するか
  def qsort_sa(n, array)
    if array.size == 0
      return array
    end
    if array.size == 1
      return array
    end

    pivot = array.sample
    before = [] # 辞書順が前
    same = [] # n文字目が同じ
    after = [] # 辞書順が後ろ

    array.each do |index|
      if @str.at_banpei(n + index) < @str.at_banpei(n + pivot)
        before.push(index)
      elsif @str.at_banpei(n + index) == @str.at_banpei(n + pivot)
        same.push(index)
      else
        after.push(index)
      end
    end

    qsort_sa(n, before).concat(qsort_sa(n + 1, same), qsort_sa(n, after))
  end
end

p SuffixArray.new("abracadabra").sa
```

結局、「ある文字列を基準に 3 つに分類して、1 つ目と 3 つ目はもう一度ソートし、2 つ目は比較する文字の位置を 1 つ進めてソートする」という処理を再帰的にやっているだけです。

3.6 おわりに

時間が足りず大変雑な記事になってしまいました。編集者として部誌の校正とか記事の整形とか LaTeX との格闘とかもしないといけなかったし、ウェブサイトの管理者として文化祭ページを更新しないといけなかったし、仕方ないですね。はい。

文字列のアルゴリズムは大変奥が深いです。この記事を読んで、ちょっとでも楽しいと思っていただけたら、ぜひ自分で学んでみてください。