Building Suffix Array: DC3 O(N) Algorithm Implementation

1. 서론

대규모의 dna 문자열을 분석하기 위해 sufffix array 를 O(N) 시간복잡도의 DC3 알고리즘으로 생성한다. DC3 알고리즘을 구현하고 이를 통해 생성한 suffix array 가 올바른지 확인한다. 또한 수행 시간을 측정하여 O(N^2)의 naive implementation 과 비교해 본다.

2. 구현

2.1. naive implementation

```
tup = lambda f: lambda argtup: f(*argtup)

def suffix_seq(s):
    for i in range(len(s)):
        yield s[i:]

def naive_suffix_map(s):
    return sorted(
        enumerate(suffix_seq(s)),
        key=tup(lambda i,k: [k,i]))

[x for x,_ in naive_suffix_map(s)] #Usage
```

문자열을 하나씩 지나면서 suffix sequence (길이 N)를 만들고 정렬한 뒤 인덱스를 남긴다. 문자열 비교가 O(N)이므로 O(N^2) 알고리즘이다. 그러나 실제로는 N 이 작을 때 나쁘지 않은 성능을 보였다.

2.2. DC3 implementation

```
def int_map(alphabet):
    return dict((s,i) for i,s in enumerate(sorted(alphabet)))
```

문자를 입력받은 alphabet 에서의 정수 rank 로의 맵핑을 반환한다. 예를 들어 alphabet 이 'abc'이면 a => 0, b => 1, c => 2 로의 매핑을 반환한다.

```
def pad3x(ints, padval=0):
    num_pad = (3 - len(ints) % 3) % 3
    return ints + [padval] * num_pad
```

입력받은 정수의 리스트 ints 의 길이가 3 의 배수가 되도록 0 을 padding 하여 반환한다.

```
def origin_indexes(length):
    return (
        list(range(0,length,3)),
        list(range(1,length,3)),
        list(range(2,length,3))
    )
```

S0, S1, S2 의 원본 문자열에서의 인덱스를 반환한다.

```
def suffix_array(imap, string):
    #[1] S0, S1, S2 인덱스 리스트와 reduced string 을 생성한다.
    unsorted_s0idx, s1idx, s2idx = origin_indexes(len(string))
    rs = pad3x([imap[c] for c in string]) + [0,0,0] #reduced string
    s1s2 = s1idx + s2idx

    #[2] S1, S2 만으로 Suffix Array 를 생성한다.
    # 만일 이후 rank 가 중복되지 않는다면 전체 SA 에 S0 와 함께 merge 된다.
    s12_sa = sorted(s1s2, key=lambda i: rs[i:i+3])
    # It could be incorrect SA if some ranks would be duplicated.
```

```
#[3] S1S2에 대한 rank를 생성하고, rank가 중복되지 않는지 확인한다.
all_unique = True
s12_rank = [0]
rank = 0
for idx0,idx1 in window(s12_sa):
    # Check 3 chars of s12_sa duplication pairwisely.
    if rs[idx0:idx0+3] == rs[idx1:idx1+3]:
        all_unique = False
    else:
        rank += 1
    s12_rank.append(rank)

s12_sa2rank = dict(zip(s12_sa, s12_rank))
#[4] 만일 rank가 중복된다면, rank가 중복되지 않을 때까지 재귀하여 rank를 생성한다.
if not all_unique:
    next_string = lmap(s12_sa2rank, s1s2)
    next_imap = int_map(set(next_string))
    s12_rank = lmap(first, sorted(
        enumerate(suffix_array(next_imap, next_string)),
        key=tup(lambda rank, sa: sa)
    ))

    # 재귀 호출의 결과로 반환된 중복 되지 않는 rank를 이용하여 S1S2를 수정한다.
    s12_sa2rank = dict(zip(s1s2, s12_rank))
    s12_sa = lmap(first, sorted(
        s12_sa2rank.items(),
        key=tup(lambda sa, rank: rank)
    ))

#[5.0] S1S2를 이용하여 S0를 정렬한다.
s0_sa = sorted(
    unsorted_s0idx,
    key=lambda i: [rs[i], s12_sa2rank.get(i+1, 0)])

#[5] 정렬된 S1S2와 S0를 합병(merge)한다.
len_s1s2 = len(s12_sa)
len_s0 = len(s0_sa)
len_sa = len_s1s2 + len_s0

suff_arr = []
i12 = i0 = 0
while i12 + i0 < len_sa:
    beg = s12_sa[i12]
    Sno = beg % 3
    beg0 = s0_sa[i0]
    # Make forms for comparison
    form12 = [*rs[beg:beg+Sno], s12_sa2rank.get(beg+Sno, 0)]
    form0 = [*rs[beg0:beg0+Sno], s12_sa2rank.get(beg0+Sno, 0)]
    assert len(form12) == len(form0), \
        f'form12 = {form12} != {form0} = form0'
    # Copy a value from s1s2 or s0
    if form12 < form0:
        suff_arr.append(beg)
        i12 += 1
    elif form12 > form0:
        suff_arr.append(beg0)
        i0 += 1
    else: assert False, 'Do not reach'
    # If cursor reaches end of s1s2/s0, append last s0/s1s2
```

```
        if i0 == len_s0:
            suff_arr = suff_arr + s12_sa[i12:]
            break
        if i12 == len_s1s2:
            suff_arr = suff_arr + s0_sa[i0:]
            break

    return suff_arr
```

2.3. Automated Correctness Test

```
from hypothesis import given, example
from hypothesis import strategies as st

from SA_DC3 import *

@given(st.text('atgc', min_size=1).map(lambda s: s + '$'))
def test_dc3(s):
    imap = int_map('atgc$')
    assert suffix_array(imap, s) == [x for x,_ in naive_suffix_map(s)]
```

자동화된 테스트를 위해서 파이썬 라이브러리 hypothesis 를 이용하여 property-based testing 을 수행하였다. 자동으로 생성된 문자열을 이용하여 DC3 알고리즘과 naive implementation 의 결과를 비교하였다.

3. 실험 및 결과

실험에서는 작성한 DC3 알고리즘과 naive implementation 의 결과가 동일한지 확인하고 각 알고리즘의 수행 시간을 측정하였다. 그림 1 은 문자열 길이 N = 10, 5010, 10010, ... 200000 으로 40 번 실험한 결과이다.

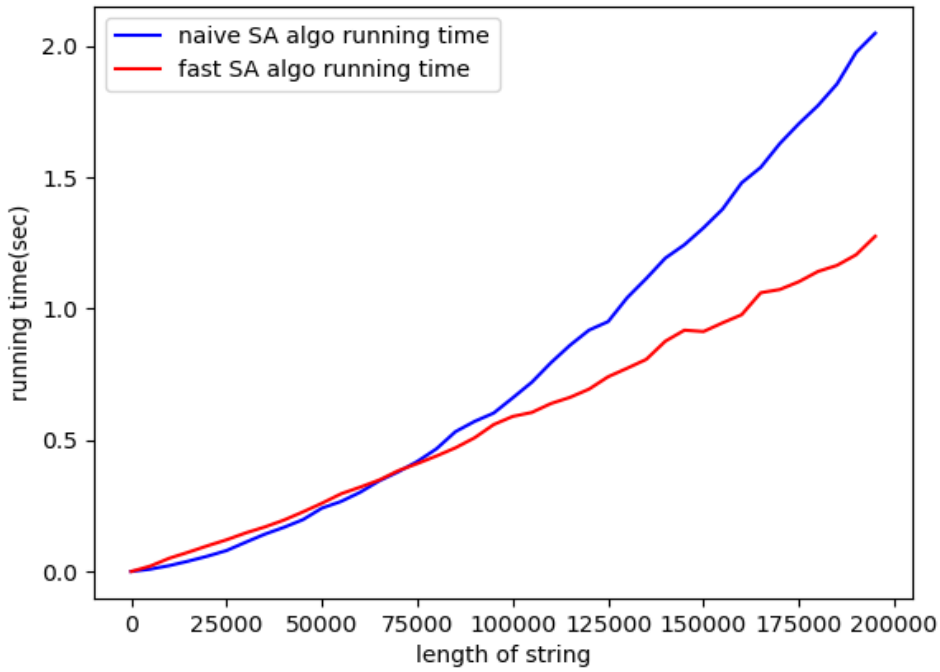

그림 1. naive implementation 과 DC3 알고리즘의 수행시간 비교

실험 결과 N 이 약 75000 개 정도일 때부터 구현한 DC3 알고리즘이 naive implementation 보다 더 나은 성능을 보이기 시작한 것을 확인할 수 있었다. 또한 DC3 구현에서 O(N logN) 정렬 알고리즘을 썼지만 O(N)에 근사한 시간 복잡도를 확인할 수 있었다.

4. 결론

DC3 알고리즘을 파이썬으로 구현하고 다양한 데이터로 검증하였다. 실험 결과 일정 크기 이하의 문자열에 대해서 DC3 는 성능상의 이점이 없다. 일정 이상의 문자열을 처리할 때 DC3 알고리즘을 적용해야 하는 것을 확인할 수 있었다.