**Building Suffix Array: DC3 O(N) Algorithm Implementation**

**1. 서론**

대규모의 dna 문자열을 분석하기 위해 sufffix array를 O(N) 시간복잡도의 DC3 알고리즘으로 생성한다. DC3 알고리즘을 구현하고 이를 통해 생성한 suffix array가 올바른지 확인한다. 또한 수행 시간을 측정하여 O(N^2)의 naive implementation과 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 N이 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 alphabet에서의 정수 rank로의 맵핑을 반환한다. 예를 들어 alphabet이 ‘abc’이면

a => 0, b => 1, c => 2로의 매핑을 반환한다.

**def** **pad3x**(ints, padval=**0**):

num\_pad = (**3** - len(ints) % **3**) % **3**

**return** ints + [padval] \* num\_pad

입력받은 정수의 리스트 ints의 길이가 3의 배수가 되도록 0을 padding하여 반환한다.

**def** **origin\_indexes**(length):

**return** (

list(range(**0**,length,**3**)),

list(range(**1**,length,**3**)),

list(range(**2**,length,**3**))

)

S0, S1, S2의 원본 문자열에서의 인덱스를 반환한다.

**def** **suffix\_array**(imap, string):

#[1] S0, S1, S2 인덱스 리스트와 reduced string을 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, S2만으로 Suffix Array를 생성한다.

# 만일 이후 rank가 중복되지 않는다면 전체 SA에 S0와 함께 merge된다.

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 hypothesis를 이용하여 property-based testing을 수행하였다. 자동으로 생성된 문자열을 이용하여 DC3 알고리즘과 naive implementation의 결과를 비교하였다.

**3. 실험 및 결과**

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

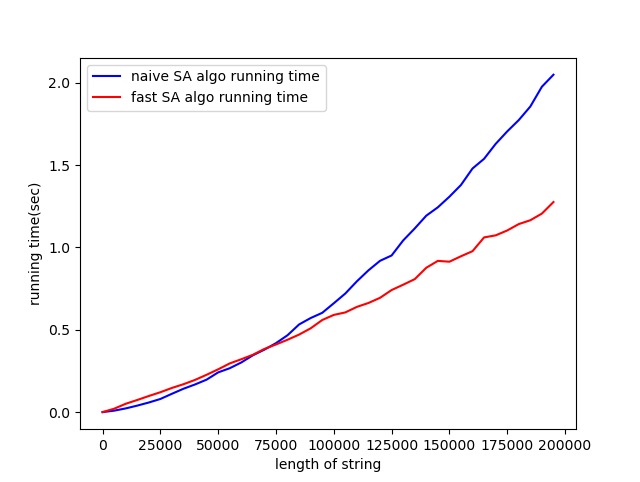


그림 1. naive implementation과 DC3 알고리즘의 수행시간 비교

실험 결과 N이 약 75000개 정도일 때부터 구현한 DC3 알고리즘이 naive implementation보다 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 DC3는 성능상의 이점이 없다. 일정 이상의 문자열을 처리할 때 DC3 알고리즘을 적용해야 하는 것을 확인할 수 있었다.