**Suffix Array Search**

**1. 서론**

큰 규모의 dna 문자열에서 주어진 패턴 문자열이 몇번 출현했는지 찾는 프로그램을 작성한다. 프로그램은 brute force를 이용하는 방법, suffix array를 이용하는 방법, longest common prefix를 이용하여 작성해보고, 3가지 방법을 비교해본다.

**2.1. brute force**

**def** **basic\_counts**(string, qstr):

occur\_idxes = set()

**for** beg **in** range(len(string)):

idx = string.find(qstr, beg)

**if** idx != -**1**:

occur\_idxes.add(idx)

**return** len(occur\_idxes)

파이썬 str 클래스의 find를 이용해서 찾는다. 겹쳐진 패턴도 찾아야하기 때문에 하나씩 밀면서 찾는다. 주어진 입력에 대해 193.415초가 걸렸다.

**2.2. suffix array**

**def** **suffix\_map**(s, suff\_arr=**None**):

**return**([s[i:] **for** i **in** suff\_arr] **if** suff\_arr

**else** sorted(suffix\_seq(s)))

**def** **bin\_find**(strs, qstr, lr):

l = **0**

r = len(strs)

**while** l + **1** != r:

m = (r - l) // **2** + l

**if** qstr < strs[m]:

r = m

**else**:

l = m

**return** r **if** lr == 'l' **else** l

입력 데이터에서 suffix array를 구성하여 찾는다. 주어진 입력에 대해서 0.063초가 걸렸다.

**2.3. suffix array with longest common prefix**

**def** **len\_lcp**(s1, s2):

length = **0**

**for** c1,c2 **in** zip(s1,s2):

**if** c1 == c2:

length += **1**

**else**:

**break**

**return** length

**def** **lcp\_find**(strs, qstr, lr):

l = **0**; r = len(strs)# - 1

lcp\_l = lcp\_r = min\_lcp = **0**

**while** l + **1** != r:

m = (r - l) // **2** + l

min\_lcp = min(lcp\_l, lcp\_r)

qstring = qstr[min\_lcp:]

string = strs[m][min\_lcp:]

**if** qstring < string:

r = m

lcp\_r = min\_lcp + len\_lcp(qstring, string)

**else**:

l = m

lcp\_l = min\_lcp + len\_lcp(qstring, string)

**return** r **if** lr == 'l' **else** l

입력 데이터에서 suffix array를 구성하고, longest common prefix를 적용하여 최적화한다. 0.217초가 걸렸다.

**3. 결론**

기본적인 알고리즘을 쓰면 193.415초, suffix array를 쓰면 0.063초, longest common prefix를 적용한 suffix array 알고리즘은 0.217초가 걸렸다. 이론적으로는 lcp를 적용할 경우 더 빨라야 하지만, 파이썬 내부 구현 문제로 인해 단순히 suffix array를 쓴 쪽이 더 빨랐다.