Дараалал дээрх паттерн илрүүлэх(Finding Patterns in Sequences)

1. Паттерн илрүүлэх

Амин бие, ялангуяа комплекс организм (хүн)–ын геномын хувьд паттерны урт, давтагдах тоо зэрэг нь паттерн илрүүлэх ач холбогдлыг нэмэгдүүлдэг.

Паттернууд нь тухайн молекулын тусгай үүргүүдээс нягт хамааралтай байдаг. Жнь:

- Онцлог паттернууд: тусгай үүрэг бүхий уургийн домайн (protein domains)
- Нуклойтид паттернууд: DNA дэд хэсгүүд

2. Тогтмол паттерн илрүүлэх гэнэ(naive) алгоритм

Даалгавар : s(N>k байх N урттай) дараалалд p (k урттай) паттернийг хайх

In [ ]:

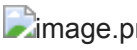
```
def search_first_occ(seq, pattern):
    found = False
    i = 0
    while i <= len(seq)-len(pattern) and not found:
        j = 0
        while j < len(pattern) and pattern[j]==seq[i+j]:
            j = j + 1
        if j== len(pattern): found = True
        else: i += 1
    if found: return i
    else: return -1

# k урттай байх s-ийн боломжит бүх дэд дарааллыг авч үзнэ
# Дэд дарааллуудын тэмдэгт бүрээр нь p-тэй харьцуулна.
# Хэрэв дэд дараалал нь бүх тэмдэгт паттернтэй тохирч байвал p-ийн тохио
# Үгүй бол дараагийн дэд дарааллыг шалгана.
# Хялбар функцууд s.find(p), s.count(p)
def search_all_occurrences(seq, pattern):
    res = []
    for i in range(len(seq)-len(pattern)+1):
        j = 0
        while j < len(pattern) and pattern[j]==seq[i+j]:
            j = j + 1
        if j == len(pattern):
            res.append(i)
    return res

def test_pat_search():
    seq = input("Input sequence: ")
    pat = input("Input pattern: ")
    print(pat, "occurs in the following positions:", )
    print( search_all_occurrences(seq, pat) )

test_pat_search()

#seqDNA = "ATAGAATAGATAATAGTC"
#print( search_first_occ(seqDNA, "GAAT") )
```

```
#print( search_first_occ(seqDNA, "TATA") )
#print( search_all_occurrences(seqDNA, "AAT") )
```

3. Хьюристик (Heuristic) алгоритм Бойер-Мур

Дараалал дээр нэгээс олон тэмдэгтээр шилжихэд чиглэсэн зарчим дээр суурилдаг.

> **Тохиромжгүй тэмдэгтийн дүрэм (Bad-character rule):** Ялгаатай тэмдэгт олдсон үед дарааллын тэмдэгт паттерн дээр олдох байрлал хүртлэх уртаар шилжүүлнэ.
>
> **Тохиромжтой залгаврын дүрэм (Good suffix rule):** Ялгаатай тэмдэгт хүртлэх хэсэг паттерн дээр олдох хүртлэх уртаар шилжинэ



In [ ]:

```
class BoyerMoore:

    def __init__(self, alphabet, pattern):
        self.alphabet = alphabet
        self.pattern = pattern
        self.preprocess()

    # Алгоритмыг үр ашигтай болгох буюу тухайн тохиолдол бүрт аль дүрмий
    # эхлүүлэхээс өмнө шаардлагатай мэдээллийг ашигтай байж болох өгөгдл
    # Шилжих урт нь зөвхөн паттернаас хамаардаг тул ямар дараалал байхаа
    # байлгахын тулд энэхүү урьдчилсан боловсруулалтыг зөвхөн паттерны х

    # bad-character rule:
    #   Dictionary төрөл ашиглана: Түлхүүр: Боломжит бүх тэмдэгт, Утга:
    #   баруун талын байрлал. Байхгүй бол -1 Шилжих уртыг хурдан тооцоол
    #   байрлал - dictionary дахь утга. Шилжих утга сөрөг байж болно, эн

    # good suffix rule ?
    #   Паттерн дээрх ялгаатай тэмдэгтийн байрлалаас хамааран шилжих урт
    def preprocess(self):
        self.process_bcr()
        self.process_gsr()

    # occ = {'A': 3, 'C': 2, 'G': -1, 'T': -1}
    def process_bcr(self):
        self.occ = {}
        for symb in self.alphabet:
            self.occ[symb] = -1
        for j in range(len(self.pattern)):
            c = self.pattern[j]
            self.occ[c] = j

    #
    def process_gsr(self):
        self.f = [0] * (len(self.pattern)+1)
        self.s = [0] * (len(self.pattern)+1)
        i = len(self.pattern)
        j = len(self.pattern)+1
        self.f[i] = j
        while i>0:
            while j<= len(self.pattern) and self.pattern[i-1] != self.pa
```

```
                if self.s[j] == 0: self.s[j] = j-i;
                j = self.f[j]
            i -= 1
            j -= 1
            self.f[i] = j
        j = self.f[0]
        for i in range(len(self.pattern)):
            if self.s[i] == 0: self.s[i] = j
            if i == j: j = self.f[j]

    def search_pattern(self, text):
        res = []
        i = 0
        while i <= len(text) - len(self.pattern):
            j= len(self.pattern)- 1
            while j>=0 and self.pattern[j]==text[j+i]: j -= 1
            if (j<0):
                res.append(i)
                i += self.s[0]
            else:
                c = text[j+i]
                i += max(self.s[j+1], j- self.occ[c])
        return res

def test():
    bm = BoyerMoore("ACTG", "ACCA")
    print (bm.search_pattern("ATAGAACCAATGAACCATGATGAACCATGGATACCCAACCAC

test()
```

4. Төгсгөлөг төлөвт детерминистик автомат (Deterministic Finite Automata)

DFA формал тодорхойлолт $M = (Q, A, q_0, \delta, F)$	• Q нь төлвийн олонлог, • A нь тэмдэгтийн цагаан толгой,	• $q_0 \in Q$ нь эхлэх төлөв, • $\delta: Q, A \rightarrow Q$ нь шилжилтийн функц, • F нь зогсох төлвийн олонлог

- Тэмдэгтүүдийг дараалллыг баруун талаас нь эхлэн уншдаг, дотоод төлвөө өөрчлөх замаар тэмдэгтүүд дээр боловсруулалт хийн ажилладаг Төгсгөлөг төлөвт автоматыг тодорхойлж болно.
  - Шинэ төлөв нь өмнөх төлөв болон уншиж байгаа тэмдэгтээс хамаардаг.
  - Өгөгдсөн паттерны хувьд боломжит цагаан толгой, төлвүүд болон шилжилтийн функцийг тодорхойлсноор DFA нь паттерн хайлтыг гүйцэтгэхэд хэрэглэж болно.
  - Q = {0, 1, … , m} ба энд m нь паттерний урт. DFA нь k гэсэн төлөвт байна гэдэг нь өмнөх байрлал дээр дараалал паттерны эхний k тэмдэгттэй тохирсон гэсэн үг.
  - q0 = 0 ба F = {m}. Зогсох төлөв нь паттерн илэрсэн байх нэг л тохиолдол байна.
  - Шилжилтийн функц (transition function)-ийг байгуулах нь бас нэг чухал алхам.

4.1 Шилжилтийн функц

$\delta(k,a) = max\_overlap(p_0 \ldots p_{k-1}\, a, p)$	• p нь паттерн, • p_i нь паттерний i дэх тэмдэгт,

- Хэрэв $k-1$ төлөвт байх үед дараалал дээрх дараагийн тэмдэгт нь паттерний k дахь тэмдэгттэй ижил бол k төлөвт шилжинэ.
- Үгүй бол ялгаатай тэмдэгт олдож, автоматаар $q_0 = 0$ төлөв рүү буцах ёстой.
  - Гэхдээ энэ нөхцлөөс өмнөх тэмдэгтүүд нь өөр нэг тохиолдолд паттернтай давхцсан байж болно.
  - Тиймээс дараалал болон паттерний эхний $k-1$ тэмдэгт давхцаж байгаа эсэхийг шалгах хэрэгтэй.
- Эдгээр дарааллын хамгийн их давхцлын урт тухайн тэмдэгтийн хувьд дараагийн төлөв болно.

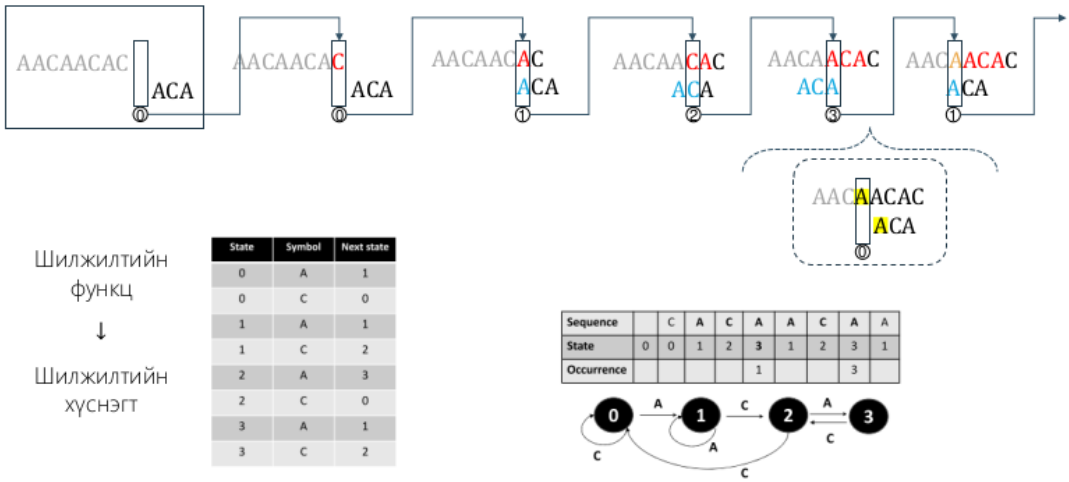
```
def overlap(s1, s2):
    maxov = min(len(s1), len(s2))
    for i in range(maxov,0,-1):
        if s1[-i:] == s2[:i]: return i
    return 0
```

s ба t хоёр дарааллын хамгийн урт давхцал:
- x-ийн хамгийн их утга
- s-ийн сүүлийн x тэмдэгтүүд t-ийн эхний x тэмдэгтүүдтэй ижил байх.

In [ ]:

```
class Automata:

    def __init__(self, alphabet, pattern):
        self.numstates = len(pattern) + 1
        self.alphabet = alphabet
        self.transition_table = {}
        self.build_transition_table(pattern)

    def build_transition_table(self, pattern):
        for q in range(self.numstates):
            for a in self.alphabet:
                prefix = pattern[0:q] + a
                self.transition_table[(q,a)] = overlap(prefix, pattern)

    def print_automata(self):
        print ("States: " , self.numstates)
        print ("Alphabet: " , self.alphabet)
        print ("Transition table:")
        for k in self.transition_table.keys():
            print (k[0], ",", k[1], " -> ", self.transition_table[k])

    def next_state(self, current, symbol):
        return self.transition_table.get((current, symbol))

    def apply_seq(self, seq):
        q = 0
        res = [q]
        for c in seq:
            q = self.next_state(q, c)
            res.append(q)
        return res

    def occurences_pattern(self, text):
        q = 0
        res = []
        for i in range(len(text)):
            q = self.next_state(q, text[i])
            if q == self.numstates-1:
                res.append(i - self.numstates + 2)
        return res

def overlap(s1, s2):
    maxov = min(len(s1), len(s2))
    for i in range(maxov,0,-1):
        if s1[-i:] == s2[:i]: return i
    return 0
```

```
def test():
    auto = Automata("ACGT", "ACA")
    auto.print_automata()
    print (auto.apply_seq("CACATGACATG"))
    print (auto.occurences_pattern("CACATGACATG"))

test()
```



Цагаан толгойн тэмдэгт A ба C бөгөөд паттерн нь "ACA" байх автомат.

5. Exercises

1. Write a Python function that, given a DNA sequence, allows to detect if there are re-peated sequences of size k (where k should be passed as an argument to the function). The result should be a dictionary with sub-sequences as keys, and their frequency as val-ues.

In [ ]:
```
# К хэмжээгээр амин хүчил гаргах
def repeats(seq, k):
    dic = {}
    for i in range(len(seq)-k+1):
        cod = seq[i:i+k]
        if cod in dic:
            dic[cod] += 1
        else: dic[cod] = 1
    res = {}
    for k in dic.keys():
        if dic[k] > 1: res[k] = dic[k]
    return res

print(repeats("ATAGAGATAGGAAGA",4))
print(repeats("ATAGAGATAGGAAGA",3))
```

2. Most introns can be recognized by their consensus sequence which is defined as: GT . . . TACTAAC . . . AC, where . . . mean an unknown number of nucleotides (between 1 and 10). Write a Python function that, given a DNA sequence, checks if it contains an in- tron, according to this definition. The result should be a list with all initial positions of the introns (empty list if there are none).

In [ ]:
```
# Finding a Motif in Dna
```

```
def searchPatt(seq, patt):
    res = []
    for i in range(len(seq) - len(patt) + 1):
      j = 0
      while j < len(patt) and patt[j] == seq[i+j]:
        j = j + 1
      if j == len(patt):
        res.append(i)
    if(res): return res
    else: return None

print(searchPatt('GATATATGCATATACTT', 'ATAT'))
```

3. In many proteins present in the membrane, there is a conserved motif that allows them to be identified in the transport process of these protein by the endosomes to be degraded in the lysosomes. This motif occurs in the last 10 positions of the protein, being character-ized by the aminoacid tyrosine (Y), followed by any two aminoacids and terminating in a hydrophobic aminoacid of the following set – phenylalanine (F), tyrosine (Y) or threo- nine (T).

a. Write a function that, given a protein (sequence of aminoacids), returns an integer
value indicating the position where the motif occurs in the sequence or −1 if it does
not occur.

In [ ]:
```
# a.
def searchPatt(s, patt):
    res = []
    seq = s[-10:] # last 10
    print(seq)
    for i in range(len(seq) - len(patt) + 1):
      j = 0
      while j < len(patt) and patt[j] == seq[i+j]:
        j = j + 1
      if j == len(patt):
        res.append(i)
    if(res): return res
    else: return -1

print(searchPatt('GATATATGCATATACTT', 'ATAT'))
```

b. Write a function that, given a list of protein sequences, returns a list of tuples, con-
taining the sequences that contain the previous motif (in the first position of the
tuple), and the position where it occurs (in the second position). Use the previous
function.

In [ ]:
```
# b.
def searchPatt(seq, patt):

    idx = []
    res = []
    for i in range(len(seq) - len(patt) + 1):
```

```
      j = 0
      while j < len(patt) and patt[j] == seq[i+j]:
        j = j + 1
      if j == len(patt):
        idx.append(i)

    for i in idx:
        res.append([seq[:i]])
        after = i+(len(patt)-1)
        res.append([seq[-after:]])

    if(res): return list(map(tuple, res))
    else: return -1

print(searchPatt('GCATATACTT', 'ATAT'))
# GCATATACTT
# before motif = G, after motif = TACTT
```

4. Write a function that given two sequences of the same length, determines if they have at most two d mismatches (d is an argument of the function). The function returns True if the number of mismatches is less or equal to d, and False otherwise. Using the pre- vious function, write another function to find all approximate matches of a pattern in a sequence. An approximate match of the pattern can have at most d characters that do not match (d is an argument of the function).

In [ ]:

```
def aprox_compare(seq1, seq2, d):
    mismatches = 0
    i = 0
    while mismatches <= d and i < len(seq1):
        if seq1[i] != seq2[i]:
            mismatches += 1
        i = i + 1
    if mismatches <= d: return True
    else: return False

def aprox_match(seq, patt='ATG', d=2):
    res = []
    # range()
    for pos in range(len(seq)-len(patt)+1):
        if aprox_compare(seq[pos:pos+len(patt)], patt, d):
            res.append(pos)
    return res

print(aprox_compare("ATGAT","ATTAA",1))
print(aprox_compare("ATGAT","ATTAA",2))
print(aprox_match('ATGATATGAT'))
```

5. Write a function that reads a file in the FASTA format and returns a list with all sequences.

In [ ]:

```
def read_Fasta (filename):
    from re import sub, search

    res = []
    sequence = None
    info = None

    fh = open(filename)
```

```
    for line in fh:
        if search(">.*", line):
                if sequence is not None and info is not None and sequenc
                    res.append(sequence)
                info = line
                sequence = ""
        else:
            if sequence is None: return None
            else: sequence += sub("\s","",line)

    if sequence is not None and info is not None and sequence != "":
                    res.append(sequence)
    fh.close()

    return res

print(read_Fasta("files/exuniprot.fasta"))
print(read_Fasta("files/NC_005816.fna"))
```

6. Files from UniProt saved in the FASTA format have a specific header structure given by:

```
db|Id|Entry Protein OS = Organism [GN = Gene] PE =
Existence SV = Version
```

Write a function that using regular expressions parses a string in this format and returns a dictionary with the different fields (the key should be the field name). Note the part in right brackets is optional, the parts in italics are the values of the fields, while the parts in upper case are constant placeholders.

In [ ]:

```
def handle_uniprot_header(header):
    from re import search

    dic = {}
    er = ">.*\|(.*)\|(\S*)(.*)OS=(.*)GN=(.*)PE=(.*)SV=(.*)"
    res =  search(er, header)
    er1 = ">.*\|(.*)\|(\S*)(.*)OS=(.*)PE=(.*)SV=(.*)"
    res1 = search(er1, header)
    if res != None:
        idprot = res.group(1)
        entry = res.group(2)
        desc = res.group(3)
        organism = res.group(4)
        gene = res.group(5)
        pe = res.group(6)
        sv = res.group(7)
    elif res1 != None:
        idprot = res1.group(1)
        entry = res1.group(2)
        desc = res1.group(3)
        organism = res1.group(4)
        gene = None
        pe = res1.group(5)
        sv = res1.group(6)
    else: return None

    dic["id"] = idprot
    dic["Entry"] = entry
```

```
    dic["Protein"] = desc
    dic["OS"] = organism
    dic["PE"] = pe
    dic["SV"] = sv
    dic["GN"] = gene

    return dic

print ( handle_uniprot_header(">sp|P27747|ACOX_RALEH Acetoin catabolism p
print ( handle_uniprot_header(">sp|P27747|ACOX_RALEH Acetoin catabolism p
```