1 Frequent Words

Funkcija ima dva parametra, `text` i `k`. Parametar `text` predstavlja nisku u kojoj tražimo podnisku dužine `k`. Povratna vrednost funkcije je skup podniski dužine `k` koje se najčešće pojavljuju.

Polazimo od praznog skupa `frequent_patterns` i praznog niza `count`. Polazeći od nulte pozicije određujemo broj pojavljivanja (funkcijom `pattern_count`) svake podniske dužine `k` i broj dodajemo u niz `count`. Zatim, određujemo najveći broj iz niza (ugrađenom funkcijom `max`). Na kraju, u skup `frequent_patterns` dodajemo sve podniske čiji je broj pojavljivanja jednak najvećem.

```
def frequent_words(text, k):
    frequent_patterns = set([])
    count = []

    for i in range(len(text) - k):
        pattern = text[i:i+k]
        count.append(pattern_count(text, pattern))

    max_count = max(count)

    for i in range(len(text) - k):
        if count[i] == max_count:
            frequent_patterns.add(text[i:i+k])

    return frequent_patterns
```

1.1 Pattern Count

Funkcija ima dva parametra, `text` i `pattern`. Prebrojava pojavljivanja zadate sekvence `pattern` u tekstu `text`.

```
def pattern_count(text, pattern):
    count = 0
    for i in range(len(text) - len(pattern)):
        if text[i:i+len(pattern)] == pattern:
            count += 1
    return count
```

1.1.1 Test primer

`text` = 'atgcgctagtcactagtgtcagttcgatgtcgat'
`k` = 3
`frequent_patterns` = ...

2 Faster Frequent Words

Funkcija ima dva parametra, `text` i `k`. Parametar `text` predstavlja nisku u kojoj tražimo podnisku dužine `k`. Povratna vrednost funkcije je par - skup podniski dužine `k` koje se najčešće pojavljuju i broj pojavljivanja sekvenci iz skupa u tekstu.

Polazi se od praznog skupa sekvenci. Prvo, određujemo niz frekvencija pojavljivanja svih podniski dužine `k` korišćenjem funkcije `computing_frequencies`. Indeks niza jedinstveno određuje nisku, i obrnuto (u tu svrhu koriste se funkcije `nuber_to_pattern` i `pattern_to_number`).

Zatim, određujemo najveću frekvenciju. Na kraju, prolazimo sve kombinacije niski nad azbukom $\{A, T, C, G\}$ i u skup dodajemo sekvence sa maksimalnom frekvencijom.

```
def faster_frequent_words(text, k):
    frequent_patterns = set([])

    frequency_array = computing_frequencies(text, k)

    max_count = max(frequency_array)

    for i in range(4**k):
        if frequency_array[i] == max_count:
            pattern = number_to_pattern(i, k)
            frequent_patterns.add(pattern)

    return (frequent_patterns, max_count)
```

2.1 Computing Frequencies

Funkcija ima dva parametra, `text` i `k`. Funkcija formira niz frekvencija pojavljivanja za sve moguće kombinacije niski nad azbukom $\{A, T, C, G\}$. Polazimo od niza nula dimenzije 4^k. Za svaku podnisku dužine `k` određujemo njen broj (funkcijom `pattern_to_number`) odnosno, indeks u nizu frekvencija, i odgovarajući element uvećamo za jedan.

```
def computing_frequencies(text, k):
    frequency_array = [0 for i in range(4**k)]

    for i in range(len(text) - k):
        pattern = text[i:i+k]
        j = pattern_to_number(pattern)
        frequency_array[j] += 1

    return frequency_array
```

2.2 Number To Pattern

Funkcija ima dva parametra, `n` - broj koji treba pretvoriti u sekvencu, i `k` - dužinu sekvence. Implementacija je rekurzivna. Izlazimo iz rekurzije kada je dužina sekvence jednaka 1, pri čemu vraćamo karakter koji odgovara trenutnoj vrednosti broja `n`. Računa se u osnovi 4. Prefiksni indeks predstavlja količnik broja `n` i broja 4. Određujemo karakter `c`, koji odgovara ostatku koji se dobija pri tom deljenju. Takođe, treba odrediti prefiksnu sekvencu koja odgovara prefiksnom indeksu, rekurzivnim pozivom, pri čemu je `k` umanjeno za 1. Vraćamo nisku koja se dobija nadovezivanjem karaktera `c` na prefiksnu sekvencu.

```
def number_to_pattern(n, k):
    if k == 1:
        return number_to_symbol(n)

    prefix_index = n // 4
    r = n % 4
    c = number_to_symbol(r)
    prefix_pattern = number_to_pattern(prefix_index, k - 1)
```

```
    return prefix_pattern + c
```

2.2.1 Pattern To Number

Funkcija ima jedan parametar, `pattern`, koji treba pretvoriti u broj. Implementacija je rekurzivna. Izlaz iz rekurzije je sekvenca dužine 0, kojoj odgovara broj 0. Broj se računa u osnovi 4, korišćenjem Hornerove sheme. Rekurzivno računamo broj prefiksa (podniska bez poslednjeg karaktera), množimo sa 4 i dodajemo broj koji odgovara poslednjem karakteru (korišćenjem funkcije `symbol_to_number`).

```
def pattern_to_number(pattern):
    if len(pattern) == 0:
        return 0

    last = pattern[-1:]
    prefix = pattern[:-1]

    return 4 * pattern_to_number(prefix) + symbol_to_number(last)
```

2.2.2 Symbol To Number

Funckija prima jedan karakter i vraća odgovarajući broj. Broj se čita iz mape koja preslikava karaktere A, T, C, G u brojeve 0, 1, 2, 3.

```
# Prevodjenje nukleotida u brojeve
def symbol_to_number(c):
    pairs = {
        'a' : 0,
        't' : 1,
        'c' : 2,
        'g' : 3
    }

    return pairs[c]
```

2.2.3 Number To Symbol

Funkcija prima jednu cifru i vraća odgovarajući karakter. Karakter se čita iz mape koja preslikava cifre 0, 1, 2, 3 u karatere A, T, C, G.

```
# Prevodjenje brojeva u nukleotide
def number_to_symbol(n):
    pairs = {
        0 : 'a',
        1 : 't',
        2 : 'c',
        3 : 'g'
    }

    return pairs[n]
```

3 Frequent Words With Mismatches

Funkcija ima tri parametra, `text`, `k` i `d` - broj dozvoljenih promašaja. Povratna vrednost je skup čestih sekvenci sa najviše `d` promašaja.

Polazimo od praznog skupa čestih sekvenci. Pravimo dva niza nula dimenzije 4^k, `close` - kanadidati za proveru i `frequency_array` - frekvencije kandidata. Za svaki uzorak dužine `k` određujemo susede - sekvence koje se od uzorka razlikuju na najviše `d` pozicija (korišćenjem funkcije `neighbors`). Za svakog suseda određujemo indeks i evidentiramo ga u nizu kandidata (niz `close`) - postavljamo mu vrednost na 1.

Prolazimo elemente niza `close` i za sve koji su evidentirani određujemo koja je sekvenca u pitanju, na osnovu indeksa (funkcijom `number_to_pattern`) i za njih se određuje broj pojavljivanja koji se pamti u nizu frekvencija (funkcijom `approximate_pattern_count`).

Zatim, određujemo najveću frekvenciju. Na kraju, u skup čestih sekvenci dodaje se svaka sekvenca čiji je frekvencija jednaka najvećoj.

```
def frequent_words_with_mismatches(text, k, d):
    frequent_patterns = set([])
    close = [0 for i in range(4**k)]
    frequency_array = [0 for i in range(4**k)]

    for i in range(len(text) - k):
        neighborhood = neighbors(text[i:i+k], d)

        for pattern in neighborhood:
            index = pattern_to_number(pattern)
            close[index] = 1

    for i in range(4**k):
        if close[i] == 1:
            pattern = number_to_pattern(i, k)
            frequency_array[i] = approximate_pattern_count(text,
                                                           pattern, d)
    max_count = max(frequency_array)

    for i in range(4**k):
        if frequency_array[i] == max_count:
            pattern = number_to_pattern(i, k)
            frequent_patterns.add(pattern)

    return frequent_patterns
```

3.1 Neighbors

Funkcija ima dva parametra, `pattern` i `d`. Povratna vrednost je skup `neighborhood`. Implementacija je rekurzivna. Prvo se proverava da li je `d` jednako 0. U tom slučaju vraćamo jednočlani skup koji sadrži samo `pattern`. To omogućava da funkciju koristimo i u slučaju kad ne želimo promašaje bez ikakvih modifikacija.

Izlazimo iz rekurzije kada je dužina uzorka jednaka 1. U tom slučaju vraćamo skup koji sadrži listu svih slova azbuke.

Pravimo skup `neighborhood` koji inicijalno sadrži praznu listu. Zatim, određujemo susede sufiksa, odnosno, sekvence bez prvog karaktera niske `pattern` i smeštamo u `suffix_neighbors`. Za svakog suseda sufiksa, koji se od sufiksa razlikuje na manje od `d` mesta, u `neighborhood` dodajemo 4 sekvence, po jedna za svako slovo azbuke, pri čemu se slovo nadovezuje na početak suseda.

Za susede koji se razlikuju na više od `d` pozicija, u `neighborhood` dodajemo suseda na čiji je početak nadovezan prvi karakter niske `pattern`, kako se razlika ne bi povećala i premašila dozvoljeni broj `d`. Kao mera za razliku koristi se Hamingovo rastojanje (funckija `hamming_distance`).

```
def neighbors(pattern, d):
    if d == 0:
        return set([pattern])

    if len(pattern) == 1:
        return set(['a', 't', 'c', 'g'])

    neighborhood = set([])

    suffix_neighbors = neighbors(pattern[1:], d)

    for text in suffix_neighbors:
        if hamming_distance(pattern[1:], text) < d:
            for x in ['a','t','c','g']:
                neighborhood.add(x + text)
        else:
            neighborhood.add(pattern[0] + text)

    return neighborhood
```

3.2 Approximate Pattern Count

Funckija ima tri parametra, `tekst`, `pattern` i `d`. Povratna vrednost je broj pojavljivanja podsekvenci u tekstu koje se od uzorka razlikuju na najvise `d` pozicija. Kao i u prethodnoj funkciji, koristi se Hamingovo rastojanje.

```
def approximate_pattern_count(text, pattern, d):
    count = 0

    for i in range(len(text) - len(pattern)):
        pattern_p = text[i:i+len(pattern)]

        if hamming_distance(pattern, pattern_p) <= d:
            count += 1

    return count
```

3.2.1 Hamming distance

Funkcija ima dva parametra, `text1` i `text2`. Povratna vrednost je brj pozicija na kojima se tekstovi razlikuju. Podrazumeva se da je dužina niski jednaka.

```
def hamming_distance(text1, text2):
    distance = 0

    for i in range(len(text1)):
        if text1[i] != text2[i]:
            distance += 1
    return distance
```

3.2.2 Test primer

`text` = 'tgactatcatcgtagtatcgatgtgcacacacgtgcgcgcgcgcgccctgtacatgatc'
`k` = 5
`d` = 2
`frequent_patterns` = ...