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` = ...

4 GC-Skew

Otvaramo fajl u FASTA formatu koji sadrži nukleotidnu sekvencu. Zanemarujemo prvi red datoteke. Dodatno, potrebno je da uklonimo beline, odnosno sve nove redove, kao i da pretvorimo slova iz velikih u mala. Nakon toga, računamo skew na osnovu nukleotida od milionitog do kraja (korišćenjem funkcije `calculate_skew`). Na kraju, crtamo skew (pomoću funkcije `draw_skew`).

```
def main():

    fajl = open('ecoli.fna', 'r')

    fajl.readline() # Zanemaruje se prvi red datoteke u FASTA formatu
    sadrzaj = ""

    sadrzaj = fajl.readlines()
    sadrzaj = "".join(sadrzaj)
    sadrzaj = sadrzaj.replace('\n', '')
    sadrzaj = sadrzaj.lower()

    skew = calculate_skew(sadrzaj[1000000:])
    draw_skew(skew)
```

4.1 Calculate Skew

Funkcija ima jedan parametar, `text`, koji predstavlja nukleotidnu sekvencu za koju računamo skew. Povratna vrednost je `skew`.

Skew je inicijalno niz nula, dimenzije datog teksta. Prolazimo sve karaktere teksta i u zavisnosti od nukleotida vršimo odgovarajuću akciju:

- ako je nukleotid `G` - prethodnu vrednost uvećavamo za jedan
- ako je nukleotid `C` - prethodnu vrednost smanjujemo za jedan
- u ostalim slučajevima ne menjamo prethodnu vrednost.

Dobijenu vrednost smeštamo u `skew` na odgovarajuću poziciju.

4.2 Draw Skew

Funkcija ima jedan parametar, `skew` - skew dijagram koji treba vizuelizovati. U tu svrhu, neophodno je uključiti pyplot iz biblioteke matplotlib.

Vrednosti na x-osi su brojevi iz intervala $[0, len(skew)]$. Pravimo `ax` koja sadrži subplot. U njoj iscrtavamo `plot` za `x` i `skew`. Možemo označiti ose (xlabel, ylabel) i postaviti naslov, pozivom funkcije `set` nad `ax`. Takođe, možemo dodati mrežu pozivom funkcije `grid` nad `ax`. Na kraju prikazujemo dijagram, pozivom funkcije `show` iz biblioteke pyplot.

```
def calculate_skew(text):
    skew = [0 for c in text]

    last = 0

    for i in range(0,len(text)):

        if text[i] == 'g':
            skew[i] = last + 1

        elif text[i] == 'c':
            skew[i] = last - 1

        else:
            skew[i] = last

        last = skew[i]

    return skew
```

```
import matplotlib.pyplot as plt

def draw_skew(skew):
    x = [i for i in range(len(skew))]

    ax = plt.subplot()
    ax.plot(x, skew)

    ax.set(xlabel='G-C', ylabel='Nucleotide',
        title='Genome GC Skew')
    ax.grid()

    plt.show()
```

5 Median String

Funkcija ima dva parametra, `dna` - niz niski koje čine nukleotidnu sekvencu, i `k` - željena dužina . Povratna vrednost je niska `median`.

Tražimo nisku dužine `k` koja je najmanje udaljena od svih iz `dna`. Uzimamo u obzir svih 4^k kombinacija i proveravamo koja najmanje udaljenda od svih. Za određivanje niske koristi se funkcija `number_to_pattern` (2.2). Zatim, za dobijenu nisku određujemo ukupno rastojanje od svih sekvenci iz `dna` (funkcijom `d`). Potrebno je još proveriti da li je to rastojanje trenutni minimum i, ako jeste, ažurirati minimalno rastojanje i nisku `meadian`.

```
def median_string(dna, k):
    distance = float('inf')
    median = ''

    for i in range(4**k):
        pattern = number_to_pattern(i, k)

        current_distance = d(pattern, dna)

        if distance > current_distance:
            distance = current_distance
            median = pattern

    return median
```

5.1 D

Funkcija ima dva parametra, `pattern` - kandidat za median, i `dna` - niz niski koje čine nukleotidnu sekvencu. Povratna vrednost je suma Hamingovih rastojanja između niske `pattern` i svih sekvenci iz `dna`.

Prolazimo jednu po jednu sekvencu iz `dna` i za svaku određujemo najmanje Hamingovo rastojanje u odnosu na `pattern`. Ideja je da u toj sekvenci tražimo podsekvencu, dužine niske `pattern` (u kodu, to je vrednost `k`), koja se najbolje poklapa sa niskom `pattern` odnosno, koja ima najmanje hamingovo rastojanje (3.2.1) u odnosu na nisku `pattern`. Nakon što je određeno najmanje rastojanje za jednu sekvencu, ono se dodaje na ukupan zbir rastojanja, što je povratna vrednost ove funkcije.

```
def d(pattern, dna):
    k = len(pattern)
    distance = 0

    for dna_string in dna:
        h_dist = float('inf')

        for i in range(len(dna_string) - k + 1):
            pattern_p = dna_string[i:i+k]
            dist = hamming_distance(pattern_p, pattern)

            if dist < h_dist:
                h_dist = dist

        distance += h_dist
    return distance
```

5.1.1 Test primer

`dna` = [
'GTAGATGTCATTAGCATGCAC',
'CCTAGCCACTCTGCCATGTCG',
'AACTCGTGCATTCTACGACTG',
'AAACTTTCCGGATCTTCATAC',
'CTACATCATCGAAGGCTACGC'
]
`k` = 4
`median` =

6 Greedy Motif Search

Funkcija ima tri parametra, `dna` - niz sekvenci, `k` - dužina motiva, i `t` - broj sekvenci u `dna`. Povratna vrednost je skup najboljih motiva.

Inicijalno, `best_motifs` sadrži prefikse svih niski iz `dna` dužine `k`. Pored najboljih motiva, pamtimo i trenutno nabolji (najmanji) skor, koji se računa korišćenjem funkcije `score`.

Polazimo od prve sekvence u nizu `dna` iz koje ćemo izdvajati podniske dužine `k`. Čuvamo skup motiva, a prvi motiv biće podniska dužine `k` u tekućoj iteraciji. Dakle, indeksiramo prvu nisku i u svakoj iteraciji izdvajamo podniske dužine `k`. Pored toga, u svakoj iteraciji iz preostalih `t` - 1 sekvenci izdvajamo podniske dužine `k` koje su najverovatnije (to određuje funkcija `most_probable_kmer`) u odnosu na matricu `profile`. Određujemo profil na osnovu motiva iz prethodnih iteracija (a dobijamo ga kao povratnu vrednost funkcije `profile`), a onda i najverovatniju podnisku. Najverovatniju podnisku dodajemo u skup motiva i on se koristi u narednoj iteraciji za pronalaženje sledećeg motiva. Kada su određeni svi motivi u tekućoj iteraciji, računa se trenutni skor i, ukoliko je bolji od trenutnog najboljeg, ažuriramo najbolji skor i motive.

```
def greedy_motif_search(dna, k, t):
    best_motifs = [dna_string[0:k] for dna_string in dna]
    best_score = score(best_motifs, k)
    first_string = dna[0]

    for i in range(len(first_string) - k):
        motifs = []
        motifs.append(first_string[i:i+k])

        for j in range(1, t):
            profile = profile_from_motifs(motifs, k, j)
            motifs.append(most_probable_k_mer(dna[j], profile, k))

        current_score = score(motifs, k)

        if current_score < best_score:
            best_motifs = copy.deepcopy(motifs)
            best_score = current_score

    return best_motifs
```

6.1 Score

Funkcija ima dva parametra, `motifs` - skup motiva, i `k` - dužina svakog od motiva. Povratna vrednost je ukupan skor. Da se podsetimo, skor predstavlja ukupan broj nepopularnih nukleotida

po kolonama.

Promenljiva `t` pamti broj motiva, a povratna vrednost biće smeštena u promenljivu `total_score`. Dakle, krećemo se po kolonama, tako da će granica za izvršavanje petlje biti vrednost `k`, što je dužina svakog od motiva. Za svaku kolonu pamtimo koliko se koji nukleotid pojavio u nizu `counts` dimenzije 4. Zatim prolazimo redom karaktere motiva iz odgovarajuće kolone i ažuriramo niz. Koristimo funkciju `symbol_to_number` (2.2.2) za dobijanje indeksa, a potom sledi jednostavna inkrementacija elementa niza na odgovarajućoj poziciji. Kada su obrađeni svi motivi, potrebno je odrediti indeks maksimuma. Na kraju, uvećati `total_score` za razliku ukupnog broja nukleotida i broj pojavljivanja najpopularnijeg nukleotida.

```
# Izracunavanje ukupnog skora za skup motiva
def score(motifs, k):
    t = len(motifs)

    total_score = 0

    for j in range(k):

        counts = [0, 0, 0, 0]

        for i in range(t):
            c = motifs[i][j]
            index = symbol_to_number(c)
            counts[index] += 1

        max_index = 0

        for i in range(1,4):
            if counts[i] > counts[max_index]:
                max_index = i

        total_score += t - counts[max_index]

    return total_score
```

6.2 Profile From Motifs

Funkcija ima dva parametra, `motifs` - skup motiva, i `k` - dužina svakog motiva. Povratna vrednost funkcije je matrica profila dimenzija $4 \times k$. Podsetimo se da vrednosti u matrici profila predstavljaju verovatnoću da se pojavi odgovarajući nukleotid na odgovarajućem mestu (ako bacamo četvorostranu pristrasnu kockicu, ove vrednosti bi bile verovatnoće da padne odgovarajući nukleotid). U ovoj implementaciji primenjeno je i Laplasovo pravilo, kako bi se izbegla vreovatnoća jednaka nuli.

Zbog Laplasovog pravila, matrica profila inicijalno je popunjena jedinicama. U prvom prolazu dvostruke petlje, koja se prvo kreće po kolonama a onda po vrstama, određujemo indeks nukleotida, a onda uvećamo odgovarajuće polje u matrici profila. U drugom prolazu dvostruke petlje, delimo svaki element matrice brojem motiva uvećanog za 2.

```
def profile_from_motifs(motifs, k, t):
    profile = [[1 for i in range(k)] for x in range(4)]

    for j in range(k):
        for i in range(t):
```

```
            index = symbol_to_number(motifs[i][j])
            profile[index][j] += 1

    for j in range(k):
        for i in range(4):
            profile[i][j] /= (t+2)

    return profile
```

6.3 Most Probable Kmer

Funkcija ima tri parametra, `dna_string`, `profile` i `k`. Povratna vrednost funkcije je najverovatnija podniska niske `dna_string` dužine `k` u odnosu na amtricu profila.

Iniciijalno, `best_kmer` je prazna niska, a `best_probability` je negativna vrednost. Prolazimo sve podniske dužine `k` i za svaku računamo verovatnoću (funkcijom `probability`). Ukoliko je ta verovatnoća veća od trenutno najbolje, ažuriramo najbolju podnisku i verovatnoću.

```
def most_probable_k_mer(dna_string, profile, k):

    best_k_mer = ''
    best_probability = -1

    for i in range(len(dna_string) - k + 1):
        pattern = dna_string[i:i+k]
        pattern_prob = probability(pattern, profile)

        if pattern_prob > best_probability:
            best_probability = pattern_prob
            best_k_mer = pattern

    return best_k_mer
```

6.3.1 Probability

Funkcija ima dva parametra, `pattern` i `profile`. Funkcija vraća verovatnoću pojave `pattern` sekvence u odnosu na zadati profil. Ukupna verovanoća dobija se kao proizvod odgovarajućih vrednosti iz matrice profil.

Inicijalno, verovatnoća je jednaka 1. Za svaki karakter iz sekvence `pattern`, određujemo njegov indeks funkcijom `symbol_to_number` (2.2.2). Zatim, trenutnu verovatnoću množimo vrednošću iz matrice profil koja odgovara datom karakteru i poziciji na kojoj se nalazi u niski.

```
def probability(pattern, profile):
    prob = 1

    for j in range(len(pattern)):
        c = pattern[j]
        index = symbol_to_number(c)

        prob *= profile[index][j]

    return prob
```

6.3.2 Test primer

```
dna = [
'GTAGATGTCATTAGCATGCAC',
'CCTAGCCACTCTGCCATGTCG',
'AACTCGTGCATTCTACGACTG',
'AAACTTTCCGGATCTTCATAC',
'CTACATCATCGAAGGCTACGC'
]
k = 4
t = len(dna)
best_motifs =
```

7 Randomized Motif Search

Funkcija ima tri parametra, `dna`, `k` i `t`. Povratna vrednost je skup najboljih motiva.

Polazimo od slučajno odabranih motiva (dobijamo ih funkcijom `random_k_mers`). Pamtimo trenutno najbolje motive (na početku su to ovi slučajno odabrani) i njihov skor (koji računamo korišćenjem funkcije `score`, 6.1).

Vrtimo petlju dok se skor popravlja. Prvo, formiramo profil od tekućih motiva (funkcijom `profile_from_motifs`, 6.2), a onda obrnuto, određujemo motive od dobijenog profila (funkcijom `motifs_from_profile`). Računamo skor novih motiva, i ako je skor manji od najboljeg, ažuriramo najbolje motive i njihov skor. Ukoliko je novi skor lošiji, tada prekidamo petlju i vraćamo najbolje motive.

```
import copy

def randomized_motif_search(dna, k, t):

    motifs = random_k_mers(dna, k, t)
    best_motifs = copy.deepcopy(motifs)
    best_score = score(best_motifs, k)

    while True:
        profile = profile_from_motifs(motifs, k, t)
        motifs = motifs_from_profile(profile, dna)
        current_score = score(motifs, k)

        if current_score < best_score:
            best_score = current_score
            best_motifs = copy.deepcopy(motifs)
        else:
            return best_motifs
```

7.1 Random Kmers

Funkcija ima tri parametra, `dna`, `k` i `t`. Povratna vrednost je skup slučajno odabranih motiva.

Skup motiva na početku je prazan. Za svaku sekvencu iz `dna` generišemo jedan sluačajan broj koji predstavlja poziciju od koje počinje motiv, a biće dužine `k`. Odabranu podsekvencu dodajemo u skup motiva.

```
def random_k_mers(dna, k, t):
    k_mers = []
```

```
    for i in range(t):
        start = random.randrange(0, len(dna[i]) - k + 1)
        dna_string = dna[i]
        k_mers.append(dna_string[start:start+k])

    return k_mers
```

7.2 Motifs From Profile

Funkcija ima dva parametra, `profile` i `dna`. Povratna vrednost je skup motiva.

Polazi se od praznog skupa motiva. Za svaku sekvencu iz `dna` određujemo najverovatniju podnisku (korišćenjem funkcije `most_probable_kmer`, 6.3) i dodajemo je u skup motiva.

```
def motifs_from_profile(profile, dna):
    motifs = []
    k = len(profile[0])

    for dna_string in dna:
        motifs.append(most_probable_k_mer(dna_string, profile, k))

    return motifs
```

7.2.1 Test primer

`dna` = [
'GTAGATGTCATTAGCATGCAC',
'CCTAGCCACTCTGCCATGTCG',
'AACTCGTGCATTCTACGACTG',
'AAACTTTCCGGATCTTCATAC',
'CTACATCATCGAAGGCTACGC'
]
`k` = 4
`t` = len(dna)
`best_motifs` =

8 Gibbs Sampler

Funkcija ima četiri parametra, `dna`, `k`, `t` i `N` - broj iteracija. Povratna vrednost je skup najboljih motiva.

Polazimo od slučajno odabranih motiva (dobijamo ih funkcijom `random_k_mers`). Pamtimo trenutno najbolje motive (na početku su to ovi slučajno odabrani) i njihov skor (koji računamo korišćenjem funkcije `score`, 6.1).

Funkcija se izvršava u `N` iteracija, a u svakoj prvo biramo slučajan broj `i` iz skupa $[0, t)$ (to je slučajno odabrani motiv koji brišemo). Koristimo pomoćnu promenljivu `selected_motifs` u koju kopiramo elemente iz `motifs`, a onda brišemo onaj na poziciji `i`, koja je slučano odabrana. Zatim, pravimo matricu profila (`profile_from_motifs`, 6.2). Na osnovu dobijenog profila, određujemo najverovatniji motiv (funkcijom `most_probable_kmer`, 6.3) i smeštamo u `motifs` na prehodno odabranu poziciju `i`, a brišemo ceo `selected_motifs`.

Nakon toga, određuje se skor novog skupa motiva i, ukoliko je manji, ažuriramo najbolje motive i njihov skor.

```
# Pronalazenje skupa motiva nakon N iteracija koriscenjem Gibs sampler-a
def gibbs_sampler(dna, k, t, N):
    motifs = random_k_mers(dna, k, t)
    best_motifs = copy.deepcopy(motifs)
    best_score = score(best_motifs, k)

    for j in range(N):
        i = random.randrange(0,t)

        selected_motifs = copy.deepcopy(motifs)
        del selected_motifs[i]

        profile = profile_from_motifs(selected_motifs, k, t-1)

        motifs[i] = most_probable_k_mer(dna[i], profile, k)
        del selected_motifs

        current_score = score(motifs, k)

        if current_score < best_score:
            best_motifs = copy.deepcopy(motifs)
            best_score = current_score

    return best_motifs
```

8.0.1 Test primer

```
dna = [
'GTAGATGTCATTAGCATGCAC',
'CCTAGCCACTCTGCCATGTCG',
'AACTCGTGCATTCTACGACTG',
'AAACTTTCCGGATCTTCATAC',
'CTACATCATCGAAGGCTACGC'
]
k = 4
t = len(dna)
N = 500
best_motifs =
```

9 Maximal Non Branching Path

Funkcija ima jedan parametar, `G` - de Bruijn graf. Funkcija vraća maksimalne nerazgranate putanje u grafu.

Putanja je inicijalno prazna lista, a dodatno održavamo mapu posećenih čvorova, koja je inicijalno prazna.

Za svaki čvor `v` u grafu `G` računamo ulazni i izlazni stepen (funkcijom `degree`). Ukoliko je jedan od ta dva različit od 1, obeležavamo da je čvor posećen. Dodatno, ukoliko je izlazni stepen veći od 0, želimo da obiđemo sve susede. Za svaki čvor `w` iz liste suseda čvora `v`, pravimo putanju koja na početku sadrži samo granu (v, w). Zatim, obeležimo da je čvor `w` posećen i računamo njegov ulazni izlazni stepen.

Pošto nam je potreban nerazgranati put, pratimo čvorove koji imaju tačno jednu ulaznu i jednu izlaznu granu. Tako, dokle god su i ulazni i izlazni stepeni čvora `w` jednaki 1, određujemo čvor `u`, čvor koji se nalazi na drugom kraju izlazne grane iz čvora `w`. Tu granu treba dodati u trenutni put. Zatim, čvor `w` dobija vrednost čvora `u` i potrebno je ponovo označiti čvor `w` kao posećen i ažurirate stepene. Kada nema više čvorova koji ispunjavaju uslov, u listu putanja dodajemo trenutni put.

Još jednom prolazimo sve čvorove grafa `G`, i za svaki čvor `v` koji nije posećen, odredićemo izlovoani ciklus (funkcijom `isloated_cycle`). Ukoliko takav ciklus postoji, dodajemo ga u listu putanja.

```
# Pronalazenje maksimalnih nerazgranatih putanja u grafu
def maximal_non_branching_paths(G):
    paths = []
    visited = {}

    for v in G:
        (v_in_deg, v_out_deg) = degree(G, v)
        if v_in_deg != 1 or v_out_deg != 1:
            visited[v] = True

            if v_out_deg > 0:
                for w in G[v]:
                    non_branching_path = [(v,w)]
                    visited[w] = True
                    (w_in_deg, w_out_deg) = degree(G, w)

                    while w_in_deg == 1 and w_out_deg == 1:
                        u = G[w][0]
                        non_branching_path.append((w,u))
                        w = u
                        visited[w] = True
                        (w_in_deg, w_out_deg) = degree(G, w)

                    paths.append(non_branching_path)

    for v in G:
        if v not in visited:
            c = isolated_cycle(G, v)
            if c != None:
                paths.append(c)

    return paths
```

9.1 Degree

Funkcija ima dva parametra, `G` - de Bruijn graf, i `v` - čvor čiji ulazni i izlazni stepen treba odrediti. Povratna vrednost je par ulazni i izlazni stepen čvora `v`.

Pošto je graf predstavljen kao mapa, koja preslikava čvorove u listu čvorova sa kojim je povezan, izlazni stepen biće dužina liste čvora `v` u mapi `G`. Ulazni stepen je broj čvorova koji u svojoj listi sadrže čvor `v`.

```
def degree(G, v):
    out_deg = len(G[v])
    in_deg = 0

    for u in G:
        if v in G[u]:
            in_deg += 1

    return (in_deg, out_deg)
```

9.2 Isolated Cycle

Funkcija ima dva parametra, `G` - de Bruijn graf, i `v` - izolovani čvor za koji želimo da nađemo ciklus. Povratna vrednost je ciklus, ako postoji, a u suprotnom `None`.

Ciklus je inicijalno prazna lista grana (odnoso, parova ulaznih i izlaznih čvora). Pošto tražimo nerazgranate puteve, i ovde je bitno da svi čvorovi imaju ulazne i izlazne stepene jednake 1. Tako da, prvi korak je određivanje stepena čvora `v` (funkcijom `degree`, 9.1). Petlja se vrti dok su ulazni i izlazni stepen jednaki 1. Biramo čvor `u` koji se nalazi na drugom kraju jedine izlazne grane čvora `v` i dodajemo tu granu u ciklus.

Ukoliko su ulazni čvor prve grane i izlazni čvor poslednje grane ciklusa jednaki, znači da smo napravili ciklus i, pritom, obišli sve čvorove u toj komponenti povezanosti grafa `G`, pa vraćamo ciklus. U suprotnom, čvor `v` dobija vrednost čvora `u`. Ponovo računamo stepen čvora `v`. Ukoliko se naiđe na čvor koji ne ispunjava uslove petlje, vratiti `None`.

```
def isolated_cycle(G, v):
        cycle = []

        (in_deg, out_deg) = degree(G, v)

        while in_deg == 1 and out_deg == 1:
                u = G[v][0]
                cycle.append((v,u))
                if cycle[0][0] == cycle[-1][1]:
                        return cycle

                v = u
                (in_deg, out_deg) = degree(G, v)

        return None
```

U nastavku su funkcije neophodne da bismo došli do reprezentacije niske u obliku de Bruijn grafa.

9.3 String To Kmers

Funkcija ima dva parametra, `dna_string` - nisku koju želimo da rasparčamo da delove, i `k` - dužina delova.

Polazimo od prazne liste k-grama. Polazeći od pozicije 0, uzimamo podniske dužine `k` i smeštamo u listu.

```
def string_to_k_mers(dna_string, k):
        k_mers = []

        for i in range(len(dna_string) - (k-1)):
                k_mer = dna_string[i:i+k]
                k_mers.append(k_mer)

        return k_mers
```

9.4 De Bruijn

Funkcija ima jedan parametar, `kmers` - lista k-grama. Povratna vrednost je graf predstavljen mapom.

Prolazimo listu, element po element. Za svaki k-gram izdvajamo prefiks `u`, bez poslednjeg karaktera i sufiks `v`, bez prvog karaktera. Ukoliko se `u` nalazi u grafu `G`, a čvor `v` nije u njegovoj listi, dodajemo čvor `v` u listu čvora `u`. Ako se `u` ne nalazi u grafu, onda ga dodajemo u graf, a lista inicijlano sadrži samo `v`. Ako čvor `v` nije u grafu, dodajemo ga sa praznom listom.

```
def debruijn_graph_from_k_mers(k_mers):
    G = {}

    for k_mer in k_mers:
        u = k_mer[:-1]
        v = k_mer[1:]

        if u in G:
            if v not in G[u]:
                G[u].append(v)
        else:
            G[u] = [v]

        if v not in G:
            G[v] = []

    return G
```

9.4.1 Test primer

`dna_string` = "AATCGTGACCTCAACT"
`k` = 3
`k_mers` = string_to_k_mers(dna_string, k)
`g` = debruijn_graph_from_k_mers(k_mers)
`paths` =

10 All Euler Cycles

Funkcija prima jedan parametra, graf `G`. Povratna vrednost je lista ciklusa.

Lista ciklusa na početku je prazna. Koristimo pomoćnu promenljivu, `all_graphs`, koja čuva kopiju grafa `G` u strukturi *deque*. Petlja se vrti dok ima grafova u `all_graphs`, odnosno, dok je njegova dužina veća od nula.

Izdvajamo jedan graf, `G_p` pozivom funkcije `popleft()`. U promenljivu `v_p` želimo da smestimo čvor sa ulaznim stepenom većim od jedan u grafu `G_p`. Inicijalizujemo ga sa `None` i onda pokušavamo da ga pronađemo.

Prolazimo sve čvorove grafa `G_p` i računamo njihov ulazni i izlazni stepen (funkcija `degree`, 9.1). Prvo čvor na koji naiđemo, sa ulaznim stepenom većim od 1, dodeljujemo promenljivoj `v_p` i prekidamo potragu.

Ukoliko smo našli takav čvor, odnosno, ukoliko njegova vrednost nije jednaka `None`, želimo da napravimo jednostavniji (u, v, w) bajpas graf. Da se podsetimo, potrebno je da iz grafa uklonimo grane (u, v) i (v, w) i da dodamo novi čvor x sa granama (u, x) i (x, v).

Prolazimo sve čvorove `u` od kojih postoje grane ka čvoru `v_p` u grafu `G_p`, i sve čvorove `w` do kojih postoji grana od čvora `v_p` u grafu `G_p`. Zatim, pravimo bajpas graf (funkcijom `bypass`). Ukoliko je novi graf povezan (što možemo proveriti funkcijom `is_connected`), dodajemo njegovu kopiju u `all_graphs`.

Ukoliko nismo našli odgovarajući čvor `v_p`, onda prolazimo čvorove `k` grafa `G_p` i određujemo isolovani ciklus iz svakog (funkcija `isloated_cycle`, 9.2). Ukoliko takav ciklus postoji, želimo da napravimo nisku od ciklusa (funkcijom `create_string_from_path`) i da je dodamo u listu ciklusa, ukoliko se već ne nalazi tamo.

```
def all_eulerian_cycles(G):
    all_graphs = deque([copy.deepcopy(G)])
    cycles = []

    while len(all_graphs) > 0:
        G_p = all_graphs.popleft()
        v_p = None
        for v in G_p:
            (in_deg, out_deg) = degree(G_p, v)

            if in_deg > 1:
                v_p = v
                break

        if v_p != None:
            for u in incoming(G_p, v_p):
                for w in outgoing(G_p, v_p):
                    new_graph = bypass(G_p, u, v, w)
                    if is_connected(new_graph):
                        all_graphs.append(copy.deepcopy(new_graph))
        else:
            for k in G_p:
                cycle = isolated_cycle(G_p, k)
                if cycle != None:
                    path = create_string_from_path(cycle)
                    if path not in cycles:
                        cycles.append(path);

    return cycles
```

10.1 Incoming

Funkcija ima dva parametra, graf `G` i čvor `v`. Funkcija vraća listu čvorova grafa `G` od kojih postoji grana do čvora `v`.

Dovoljno je da jednom petljom prođemo sve čvorove grafa `G` i sve koji sadrže čvor `v` u svojoj listi dodamo u listu.

```
def incoming(G, v):
    in_list = []

    for u in G:
        if v in G[u]:
            in_list.append(u)

    return in_list
```

10.2 Outgoing

Funkcija ima dva parametra, graf `G` i čvor `v`. Funkcija vraća listu čvorova do kojih postoje grane iz čvora `v` u grafu `G`, odnosno, vraća listu čvora `v`.

```
def outgoing(G, v):
    return G[v]
```

10.3 Bypass

Funkcija ima četiri parametra, graf `G` i čvorove `u`, `v` i `w`. Povratna vrednost je novi graf sa izmenjenim granama. Još jednom, treba izbaciti grane (u, v) i (v, w), dodati novi čvor x (u kodu je `v'`, kako bi implementacija funkcije za pravljenje niske bila olakšana) povezati ga sa čvorom `u` i čvorom `w`.

Prvo, kopiramo graf `G` u `G_p`, koji ćemo dalje menjati. Zatim, iz liste čvora `u` brišemo čvor `v`, a iz liste čvora `v` brišemo čvor `w`. Onda dodajemo novi čvor, `v'` u listu čvora `u`, a lista čvora `v'` sadržaće samo čvor `w`. Na kraju vraćamo izmenjeni graf.

```
def bypass(G, u, v, w):
    G_p = copy.deepcopy(G)
    G_p[u].remove(v)
    G_p[v].remove(w)
    G_p[u].append(v+"'")   #v'
    G_p[v+"'"] = [w]
    return G_p
```

10.4 Is Connected

Funkcija ima jedan parametar, graf `G`. Povratna vrednost je tipa boolean.

Održavamo mapu posećenosti. Za svaki čvor `v` u grafu `G` pozivamo pomoćnu proceduru koja će izvršiti obilazak grafa u dubinu (`DFS`) počevši iz čvora `v` nakon čega prekidamo petlju.

Za svaki čvor vu grafu `G` proveravamo da li se nalazi u mapi posećenih čvorova. Ukoliko bar jedan čvor nije u mapi, graf nije povezan i vraćamo `False`. Inače, vraćamo `True`.

```
def is_connected(G):

    visited = {};
```

```
    for v in G:
        DFS(G,v,visited)
        break

    for v in G:
        if v not in visited:
            return False

    return True
```

10.5 Create String From Path

Funkcija ima jedan parametar, `path` - putanja na osnovu koje pravimo nisku.

Niska na početku sadrži karaktere ulaznog čvora prve grane. Ostale karaktere dobijamo proslaskom kroz sve grane na putanji, i izdvajanjem poslednjeg karaktera izlaznog čvora te grane.

```
def create_string_from_path(path):

    dna_string = path[0][0].replace("'",'')

    for i in range(len(path)):
        dna_string += path[i][1].replace("'",'')[-1]

    return dna_string
```

10.5.1 DFS

Funkcija ima tri parametra, graf `G`, čvor `v` i mapu posećenosti `visited`. Nema povratne vrednosti.

Čvor `v`, iz kog kreće obilazak grafa, obeležimo da je posećen. Zatim, prolazimo sve čvorove `w`, koji se nalaze u listi čvora `v`. Za svaki čvor koji do tad nije posećen, pozivamo istu proceduru.

```
def DFS(G, v, visited):
    visited[v] = True

    for w in G[v]:
        if w not in visited:
            DFS(G, w, visited)
```

10.5.2 Test primer

```
G = {
   'AT' : ['TC'],
   'TC' : ['CG'],
   'CG': ['GA','GG'],
   'GA':['AT','AC'],
   'AC':['CG'],
   'GG':['GA']
}
cycles =
```

11 String Spelled By Gapped Patterns

Funkcija ima tri parametra, `gapped_patterns` - skup parova k-grama, `k` i `d` - rastojanje dva susedna para k-grama. Povratna vrednost je niska sačinjena od zadatih k-grama.

Razdvajamo parove u posebne liste, `first_patterns` i `second_patterns`. Zatim, formiramo prefiksne (od `first_patterns`) i sufiksne (od `second_patterns`) niske (funkcijom `string_spelled_by_patterns`).

Prefiksna i sufiksna niska imaju prefiks, odnoso, sufiks dužine `k+d`, a ostatak niski treba da im se poklopi. Tako da, prolazimo ove niske karakter po karakter. Prefiksu prolazimo počevši od pozicije `k+d` do kraja, a sufiksnu prolazimo od 0 do `len(prefiks_string)-k-d`. Ukoliko naiđemo na nepoklapanje nukleotida na nekoj poziciji, ispisujemo odgovarajuću poruku i vraćamo praznu nisku. Ako su svi nukleotidi u redu, vraćamo nisku koja se dobije nadovezivanjem sufiksa iz sufiksne niske na prefiksnu nisku.

```
def string_spelled_by_gapped_patterns(gapped_patterns, k, d):
    first_patterns = [s[0] for s in gapped_patterns]
    second_patterns = [s[1] for s in gapped_patterns]

    prefix_string = string_spelled_by_patterns(first_patterns, k)
    suffix_string = string_spelled_by_patterns(second_patterns, k)

    print(prefix_string)
    print(suffix_string)

    for i in range(k+d, len(prefix_string)):
        if prefix_string[i] != suffix_string[i-k-d]:
            print('There is no string spelled by the gapped patterns')
            return ''
    return prefix_string + suffix_string[-k-d:]
```

11.1 String Spelled By Patterns

Funkcija ima dva parametra, `patterns` - niske koje treba spojiti, i `k` - dužina svake niske. Povratna vrednost je DNK niska sastavljena od datih k-grama.

Početna vrednost niske `dna_string` je k-gram bez poslednjeg karaktera. Prolaskom svih k-grama iz liste `patterns`, gradimo nisku nadovezivanjem poslednjeg karaktera iz svakog.

```
def string_spelled_by_patterns(patterns, k):
    dna_string = patterns[0][:-1]

    for i in range(0, len(patterns)):
        dna_string += patterns[i][-1]

    return dna_string
```

11.1.1 Test primer

`gapped_patterns` = [('CTG','CTG'),('TGA','TGA'),('GAC','GAC'),('ACT','ACT')]
`k` = 3
`d` = 1
`string` =

12 Linear Spectrum

Funkcija ima tri parametra, `peptide` - peptid čiji spektar želimo da odredumo, `amino_acid` - lista aminokiselina, i `amino_acid_mass` - lista celih brojeva koji predstavljaju mase aminokiselina. Povratna vrednost je linearni spektar datog peptida.

Želimo da generišemo teorijski spektar, ali pre nego što to odredimo, prvo ćemo napraviti niz prefiksnih masa. Ovaj pristup zasnovan je na pretpostavci da je masa bilo kod potpeptida jednaka razlici masa dva prefiksa.

Na početku, lista `prefix_mass` ima samo jedan element i to 0. Dodatno, pamtimo trenutnu masu peptida u promenljivoj `current_mass`, koja je na početku jednaka 0. Redom prolazimo sve aminokiseline u peptidu i svaku moramo pronaći u nizu aminokiselina, kako bismo dobili njenu masu. U `prefix_mass` dodajemo novi element čija je vrednost jednaka trenutnoj masi uvećanoj za masu trenutne aminokiseline. Dodatno, uvećavamo trenutnu masu za masu trenutne aminokiseline.

Sada, kada imamo prefiksne mase, možemo odrediti linearni spektar. Linearni spektar predstavljen je kao lista, i na početku ima samo jedan element - 0. Redom prolazimo sve vrednosti u listi prefiksnih masa u prvoj petlji, a u drugoj petlji prolazimo istu listu od prve sledeće pozicije. U spektar dodajemo razliku veće i manje prefikse mase. Pre nego što vratimo listu, bitno je da je sortiramo i to rastuće!

```
1  def linear_spectrum(peptide, amino_acid, amino_acid_mass):
2      prefix_mass = [0]
3      current_mass = 0
4      for i in range(len(peptide)):
5          for j in range(20):
6              if amino_acid[j] == peptide[i]:
7                  prefix_mass.append(current_mass + amino_acid_mass[j])
8                  current_mass += amino_acid_mass[j]
9
10     linear_spectrum = [0]
11     for i in range(len(prefix_mass)):
12         for j in range(i+1, len(prefix_mass)):
13             linear_spectrum.append(prefix_mass[j] - prefix_mass[i])
14
15     linear_spectrum.sort()
16     return linear_spectrum
```

13 Cyclic Spectrum

Funkcija ima tri parametra, `peptide` - peptid čiji spektar želimo da odredumo, `amino_acid` - lista aminokiselina, i `amino_acid_mass` - lista celih brojeva koji predstavljaju mase aminokiselina. Povratna vrednost je ciklični spektar datog peptida.

I u ovoj funkciji nam je potrebna prefiksna masa. Na početku, lista `prefix_mass` ima samo jedan element i to 0. Dodatno, pamtimo trenutnu masu peptida u promenljivoj `current_mass`, koja je na početku jednaka 0. Redom prolazimo sve aminokiseline u peptidu i svaku moramo pronaći u nizu aminokiselina, kako bismo dobili njenu masu. U `prefix_mass` dodajemo novi element čija je vrednost jednaka trenutnoj masi uvećanoj za masu trenutne aminokiseline. Dodatno, uvećavamo trenutnu masu za masu trenutne aminokiseline.

Nakon što smo odredili listu prefiksnih masa, određujemo peptitdnu masu. Peptidna masa jednaka je poslednjoj prefiksnoj masi i njenu vrednost čuvamo u promenljivoj `peptide_mass`.

Ciklični spektar predstavljen je kao lista, i na početku ima samo jedan element - 0. Redom prolazimo sve vrednosti u listi prefiksnih masa u prvoj petlji, a u drugoj petlji prolazimo istu

listu od prve sledeće pozicije. U spektar dodajemo razliku veće i manje prefikse mase.

Razlika u odnosu na linearni spektar jeste u sledećem koraku. Ukoliko prva petlja nije u prvoj iteraciji, a druga petlja nije u poslednjoj iteraciji, onda u spektar dodajemo i vrednost peptidne mase umanjene za prethodnu razliku veće i manje prefiksne mase. I u ovom slučaju, pre nego što vratimo spektar, moramo da ga sortiramo rastuće!

```
1  def cyclic_spectrum(peptide, amino_acid, amino_acid_mass):
2      prefix_mass = [0]
3      current_mass = 0
4      for i in range(len(peptide)):
5          for j in range(20):
6              if amino_acid[j] == peptide[i]:
7                  prefix_mass.append(current_mass + amino_acid_mass[j])
8                  current_mass += amino_acid_mass[j]
9
10     peptide_mass = prefix_mass[-1]
11     cyclic_spectrum = [0]
12     for i in range(len(prefix_mass)):
13         for j in range(i+1, len(prefix_mass)):
14             cyclic_spectrum.append(prefix_mass[j] - prefix_mass[i])
15             if i > 0 and j < len(prefix_mass)-1:
16                 cyclic_spectrum.append(peptide_mass -
17                     (prefix_mass[j] - prefix_mass[i]))
18
19     cyclic_spectrum.sort()
20     return cyclic_spectrum
```

13.1 Test primer

`amino_acid` = ['G', 'A', 'S', 'P', 'V', 'T', 'C', 'I', 'L', 'N', 'D', 'K', 'Q', 'E', 'M', 'H', 'F', 'R', 'Y', 'W']
`amino_acid_mass` = [57, 71, 87, 97, 99, 101, 103, 113, 113, 114, 115, 128, 128, 129, 131, 137, 147, 156, 163, 186]
`peptide` = „NQEL”
`linear_spectrum` =
`cyclic_spectrum` =

14 Cyclopeptide Sequencing

Funkcija ima tri parametra, `spectrum` - ciklični spektar peptida, `amino_acid` - lista aminokiselina, i `amino_acid_mass` - lista masa aminokiselina. Funkcija nema povratnu vrednost, već se svi peptidi direktno ispisuju.

Polazimo od liste `peptides` koja sadrži samo jedan član - praznu nisku. Brojač `i` inicijalno ima vrednost 1. Petlju vrtimo dok `peptides` ima elemente. U svakoj iteraciji pravimo novu listu (`next_peptides`) peptida koju treba obraditi u narednoj iteraciji. Trenutnu listu peptida proširujemo svim mogućim aminokiselinama (funkcijom `expand`, 14.1), a onda je kopiramo u listu `next_peptides`.

Prolazimo sve peptide, element po element. Tražimo one peptide čija je masa (određujemo je funkcijom `mass`, 14.2) jednaka roditeljskoj masi (funkcija `parent_mass`, 14.3) i čiji je spektar jednak zadatom - `spectrum`. Svaki peptid koji zadovolji ove uslove ispisujemo na standardni izlaz. Peptid, čija se masa poklopila sa roditeljskom, brišemo iz liste za sledeću iteraciju. Peptid, koji nije zadovoljio uslov za masu, proveravamo da li je konzistentan sa spektrom (funkcijom

`consistent`, 14.4) i, ako nije, brišemo ga iz liste za sledeću iteraciju. Na kraju, `peptides` dobija vrednost liste za sledeću iteraciju, `next_peptides`.

```
def cyclopeptide_sequencing(spectrum, amino_acid, amino_acid_mass):
    peptides = ['']
    i = 1
    while len(peptides) > 0:
        next_peptides = []
        peptides = expand(peptides, amino_acid)
        next_peptides = copy.copy(peptides)
        for peptide in peptides:
            if mass(peptide, amino_acid, amino_acid_mass) == parent_mass(
    ↪ spectrum):
                if cyclic_spectrum(peptide, amino_acid, amino_acid_mass) ==
    ↪ spectrum:
                    print(peptide)
                next_peptides.remove(peptide)
            elif not consistent(peptide, spectrum, amino_acid, amino_acid_mass)
    ↪ :
                next_peptides.remove(peptide)
        peptides = next_peptides
```

14.1 Expand

Funkcija ima dva parametra, `peptides` - listu peptida koju treba proširiti, i `amino_acid` - lista aminokiselina. Povratna vrednost je lista proširenih peptida.

Lista `extension` na početku je prazna. Prolazimo redom sve peptide, i za svaki peptid prolazimo sve aminokiselina i redom nadovezujemo, jednu po jednu, i tako izmenjen peptid dodajemo u listu.

```
def expand(peptides, amino_acid):
    extension = []

    for peptide in peptides:
        for aa in amino_acid:
            extension.append(peptide + aa)

    return extension
```

14.2 Mass

Funkcija ima tri parametra, `peptide`, `amino_acid` i `amino_acid_mass`. Povratna vrednost je ukupna masa peptida.

Ukupnu masu peptida odredićemo kao sumu masa svih aminokiselina u lancu. Zbog toga, prolazimo redom sve aminokiseline u peptidu, pronalazimo je u listi `amino_acid`, kako bismo dobili njenu masu, i uvećavamo ukupnu masu za masu te aminokiseline.

```
def mass(peptide, amino_acid, amino_acid_mass):
    total_mass = 0

    for i in range(len(peptide)):
        for j in range(len(amino_acid)):
            if peptide[i] == amino_acid[j]:
```

```
                total_mass += amino_acid_mass[j]

    return total_mass
```

14.3 Parent Mass

Funkcija ima jedan parametar - ciklični spektar peptida `spectrum`. Povratna vrednost je masa peptida, odnosno, poslednja vrednost u spektru.

```
def mass(peptide, amino_acid, amino_acid_mass):
    total_mass = 0

    for i in range(len(peptide)):
        for j in range(len(amino_acid)):
            if peptide[i] == amino_acid[j]:
                total_mass += amino_acid_mass[j]

    return total_mass
```

14.4 Consistent

Funkcija ima četiri parametra, `peptide`, `target_spectrum`, `amino_acid` i `amino_acid_mass`. Povratna vrednost je tipa boolean i zavisi od toga da li je peptid konzistentan sa spektrom.

Da bismo proverili da li je peptid konzistentan sa spektrom, potrebno je da imamo njegov spektar i da ih uporedimo. Odredimo linearan spektar peptida (funkcijom `linear_spectrum`, 12) i smestimo ga u `peptide_linear_spectrum`. Nakon toga, prolazimo redom sve vrednosti u peptidnom spektru i tražimo ih u zadatom spektru. Ukoliko se makar jedna vrednost iz `peptide_linear_spectrum` ne nalazi u `target_spectrum`, vraćamo `False`. Ukoliko su sve vrednosti pronađene, vraćamo `True`.

```
def consistent(peptide, target_spectrum, amino_acid, amino_acid_mass):
    peptide_linear_spectrum = linear_spectrum(peptide, amino_acid,
    ↪ amino_acid_mass)

    for aa in peptide_linear_spectrum:
        found = False
        for aa_p in target_spectrum:
            if aa_p == aa:
                found = True
        if found == False:
            return False

    return True
```

14.4.1 Test primer

`amino_acid` = ['G', 'A', 'S', 'P', 'V', 'T', 'C', 'I', 'L', 'N', 'D', 'K', 'Q', 'E', 'M', 'H', 'F', 'R', 'Y', 'W']
`amino_acid_mass` = [57, 71, 87, 97, 99, 101, 103, 113, 113, 114, 115, 128, 128, 129, 131, 137, 147, 156, 163, 186]
`peptide` = „SPQR”
`spectrum` = = cyclic_spectrum(peptide, amino_acid, amino_acid_mass)
`ispis`:

15 Leadrboard Cyclopeptide Sequencing

Funkcija ima četiri parametra, `spectrum`, `N` - koliko najboljih peptida zadržavamo, `amino_acid`, `amino_acid_mass`. Funkcija vraća vodeći peptid.

Ideja je da se održava leaderboard kako bismo mogli da kontrolišemo broj kandidata. U svakoj iteraci zadržavamo prvih `N` peptida na tabeli, s tim da ako je `N`-ti peptid izjednačen sa jednim ili više drugih peptida, svi oni ulaze u sledeću iteraciju. Koliko je peptid dobar određujemo pomoću njihovog skora.

Krećemo od `leaderboard` sa jednim elementom, praznom niskom, a to je i početna vrednost vodećeg peptida (`leader_peptide`). Petlja se izvršava dok ima peptida na tabeli. U svakoj iteraciji pravimo novu listu (`next_leaderboard`) peptida koju treba obraditi u narednoj iteraciji. Trenutnu tabelu proširujemo svim mogućim aminokiselinama (funkcijom `expand`, 14.1), a onda je kopiramo u listu `next_leaderboard`.

Za svaki peptid na tabeli proveravamo da li mu je masa (`mass`, 14.2) jednaka roditeljskoj (`parent_mass`, 14.3). Ako jeste, proveravamo da li je njegov skor veći od skora vodećeg peptida (funkcija `score`, 15.1), i, ako jeste, ažuriramo vodeći peptid. Ako ne važi uslov jednakosti za mase, proveravamo da li je masa peptida veća od roditeljske mase i, ako jeste, uklanjamo ga iz tabele za sledeću iteraciju.

Pre nego što pređemo u sledeću iteraciju, potrebno je da skratimo tabelu, tako da zadrži najboljih `N` peptida. Ovo postižemo funkcijom `trim`, 15.2.

```
1  def leaderboard_cyclopeptide_sequencing(spectrum, N, amino_acid,
       ↪ amino_acid_mass):
2      leaderboard = ['']
3      leader_peptide = ''
4      while len(leaderboard) > 0:
5          next_leaderboard = []
6          leaderboard = expand(leaderboard, amino_acid)
7          next_leaderboard = copy.copy(leaderboard)
8          for peptide in leaderboard:
9              if mass(peptide, amino_acid, amino_acid_mass) == parent_mass(
       ↪ spectrum):
10                 if score(peptide, spectrum, amino_acid, amino_acid_mass) >
       ↪ score(leader_peptide, spectrum, amino_acid, amino_acid_mass):
11                     leader_peptide = peptide
12             elif mass(peptide, amino_acid, amino_acid_mass) > parent_mass(
       ↪ spectrum):
13                 next_leaderboard.remove(peptide)
14         leaderboard = trim(next_leaderboard, spectrum, N, amino_acid,
       ↪ amino_acid_mass)
15     return leader_peptide
```

15.1 Score

Funkcija ima četiri parametra, `spectrum_2`, `N` - koliko najboljih peptida zadržavamo, `amino_acid`, `amino_acid_mass`. Funkcija vraća broj vrednosti koje su zajedničke za spektrum datog peptida i datog spektra.

Prvo ćemo odrediti ciklični spektrum peptida i smestiti ga u `spectrum_1`. Petlja se izvršava dok se nalazimo u okviru dozovljenih vrednosti za indekse lista `spectrum_1` i `spectrum_2`. Ukoliko se vrednosti na trenutnim pozicijama poklapaju, uvećati skor i oba brojača. Ukoliko je vrednost iz prvog spektra manja od vrednosti iz drugog spektra, uvećati samo prvi brojač, u obrnutoj situaciji, uvećati samo drugi brojač.

```
def score(peptide, spectrum_2, amino_acid, amino_acid_mass):
  p1 = 0
  p2 = 0
  score = 0

  spectrum_1 = cyclic_spectrum(peptide, amino_acid, amino_acid_mass)

  while p1 < len(spectrum_1) and p2 < len(spectrum_2):
    if spectrum_1[p1] == spectrum_2[p2]:
      score += 1
      p1 += 1
      p2 += 1
    elif spectrum_1[p1] < spectrum_2[p2]:
      p1 += 1
    else:
      p2 += 1

  return score
```

15.2 Trim

Funkcija ima pet parametara, `leaderboard`, `spectrum`, `N`, `amino_acid` i `amino_acid_mass`. Funkcija vraća skraćenu tabelu.

Prvo ćemo odrediti skor za svaki peptid na tabeli. Prolazimo redom sve peptide u `leaderboard` i u listu `linear_scores` (koja je inicijalno prazna) dodajemo linearni skor trenutnog peptida (funkcija `linear_score`, 15.2.1).

Pravimo novu listu koja sadrži zipovane vrednosti iz `linear_scores` i vrednosti iz tabele. Zatim, listu sortiramo u opadajućem poretku. Sada, `leaderboard` možemo ažurirati tako što iskopiramo samo drugi element zipovanog para.

Prolazimo elemente liste sa zipovanim vrednostima, počevši od pozicije `N` do kraja. Ako je skor j-tog elementa strogo manji od skora `N-1`-vog elementa, onda skraćujemo tabelu do pozicije j, odnosno, u `leaderboard` kopiramo drugi element para prvih j elemenata liste `leaderboard_zipped` i vraćamo `leaderboard`. Ovaj korak je bitan jer se može desiti da su elementi posle `N`-tog imaju isti skor kao i taj. Nakon petlje vratiti ceo `leaderboard` jer nije ništa moglo da se odseče.

```
def trim(leaderboard, spectrum, N, amino_acid, amino_acid_mass):
  linear_scores = []
  for j in range(len(leaderboard)):
    peptide = leaderboard[j]
    linear_scores.append(linear_score(peptide, spectrum, amino_acid,
    ↪ amino_acid_mass))

  leaderboard_zipped = list(zip(linear_scores, leaderboard))
  leaderboard_zipped.sort(reverse=True)

  leaderboard = [el[1] for el in leaderboard_zipped]
  for j in range(N, len(leaderboard_zipped)):
    if leaderboard_zipped[j][0] < leaderboard_zipped[N-1][0]:
      leaderboard = [el[1] for el in leaderboard_zipped[:j]]
      return leaderboard
  return leaderboard
```

15.2.1 Linear Score

Funkcija ima četiri parametra, `spectrum_2`, `N` - koliko najboljih peptida zadržavamo, `amino_acid`, `amino_acid_mass`. Funkcija vraća broj vrednosti koje su zajedničke za spektrum datog peptida i datog spektra.

Prvo ćemo odrediti linear spektrum peptida i smestiti ga u `spectrum_1`. Petlja se izvršava dok se nalazimo u okviru dozovljenih vrednosti za indekse lista `spectrum_1` i `spectrum_2`. Ukoliko se vrednosti na trenutnim pozicijama poklapaju, uvećati skor i oba brojača. Ukoliko je vrednost iz prvog spektra manja od vrednosti iz drugog spektra, uvećati samo prvi brojač, u obrnutoj situaciji, uvećati samo drugi brojač.

```
1  def linear_score(peptide, spectrum_2, amino_acid, amino_acid_mass):
2    p1 = 0
3    p2 = 0
4    score = 0
5
6    spectrum_1 = linear_spectrum(peptide, amino_acid, amino_acid_mass)
7
8    while p1 < len(spectrum_1) and p2 < len(spectrum_2):
9      if spectrum_1[p1] == spectrum_2[p2]:
10       score += 1
11       p1 += 1
12       p2 += 1
13     elif spectrum_1[p1] < spectrum_2[p2]:
14       p1 += 1
15     else:
16       p2 += 1
17
18   return score
```

15.2.2 Test primer

`amino_acid` = ['G', 'A', 'S', 'P', 'V', 'T', 'C', 'I', 'L', 'N', 'D', 'K', 'Q', 'E', 'M', 'H', 'F', 'R', 'Y', 'W']
`amino_acid_mass` = [57, 71, 87, 97, 99, 101, 103, 113, 113, 114, 115, 128, 128, 129, 131, 137, 147, 156, 163, 186]
`peptide` = ,,SPQR"
`spectrum` = = cyclic_spectrum(peptide, amino_acid, amino_acid_mass)
`N` = 10
`ispis`:

16 Manhattan Tourist

Funkcija ima četiri parametra, `n`, `m`, `down` - matrica cene grana koje vode dole, i `right` - cene grana koje vode desno. Funkcija vraća najveću cenu puta od `(0, 0)` do `(n-1, m-1)`.

Održavamo matricu `s`, dimenzija $n \times m$, koja čuva cenu puta do svakog čvora. Svi na početku imaju cenu 0. Dodatno, čuvamo matricu `backtrack`, dimenzije $n \times m$, u kojoj za svaki čvor pamtimo prethodnika, odnosno, čvor iz kog smo došli do tog. Čvorovi su predstavljeni kao par vrednosti indeksa u matrici. Prilikom inicijalizacije matrice, svim čvorovima postaviti čvor (0, 0) za prethodnika.

Prvo ćemo se pozabaviti izračunavanjem cena i prethodnika čvorova za koje nemamo izbor, odnosno, do kojih postoji put samo iz jednog čvora. To su elementi prve vrste i prve kolone. Prethodnika za element na poziciji (0, 0) moramo posebno označiti jer iz njega krećemo. Najbolje je staviti neke nevažeće vrednosti za indekse, kao na primer (-1, -1).

Za svaki element prve kolone, cenu računamo kao zbir cene elementa iznad njega i cene na njegovoj poziciji iz matrice `down`. Prethodnik je element iznad njega jer smo njegovu cenu iskoristili. Za svaki element prve vrste, cenu računamo kao zbir cene elementa levo od njega i cene na njegovoj poziciji iz matrice `right`. Prethodnik je element levo od njega jer smo njegovu cenu iskoristili. Obe petlje započeti od 1 jer je element (0, 0) već inicijalizovan i ne upada u ove šablone!

Ostale elemente obradićemo redom, kroz dvostruku petlju. Za svaki element računamo cenu dolaska odozgo (`to_down`), kao zbir cene elementa iznad njega i cene na njegovoj poziciji iz matrice `down`, i cenu dolaska sleva (`to_right`), kao zbir cene elementa levo od njega i cene na njegovoj poziciji iz matrice `right`. Biramo veću cenu, zapisujemo je u matricu cena i u matrici `backtrack` upisujemo odgovarajući čvor za prethodnika.

Kada su određene vrednosti za sve elemente, želimo da ispišemo čitavu putanju (unazad, od `(n-1, m-1)` do `(0, 0)`), nakon čega vraćamo vrednost elementa `n-1, m-1` matrice cena. Polazimo od poslednjeg elementa, `(n-1, m-1)`, i vrtimo petlju dok u matrici `backtrack` ne naiđemo na element čija je vrednost jednaka (-1, -1) - podsetimo se da je to pogodno izabran prethodnik za početni čvor.

```
1  def manhattan_tourist(n, m, down, right):
2    s = [[0 for j in range(m)] for i in range(n)]
3
4    backtrack = [[(0,0) for j in range(m)] for i in range(n)]
5
6    backtrack[0][0] = (-1, -1)
7
8    for i in range(1,n):
9      s[i][0] = s[i-1][0] + down[i][0]
10     backtrack[i][0] = (i-1, 0)
11
12   for j in range(1,m):
13     s[0][j] = s[0][j-1] + right[0][j]
14     backtrack[0][j] = (0, j-1)
15
16   for i in range(1, n):
17     for j in range(1, m):
18
19       to_down = s[i-1][j] + down[i][j]
20       to_right = s[i][j-1] + right[i][j]
21
22       if to_down > to_right:
23         backtrack[i][j] = (i-1, j)
```

```
24          s[i][j] = to_down
25        else:
26          backtrack[i][j] = (i, j-1)
27          s[i][j] = to_right
28
29    i = n-1
30    j = m-1
31    while backtrack[i][j] != (-1,-1):
32      print(backtrack[i][j])
33      i = backtrack[i][j][0]
34      j = backtrack[i][j][1]
35
36    return s[n-1][m-1]
```

16.1 Test primer

`n` = 3
`m` = 3
`down` = [
    [0, 0, 0, 0],
    [0, 1, 2, 1],
    [0, 1, 1, 1],
    [0, 1, 1, 1]
]

`right` = [
    [0, 0, 0, 1],
    [0, 3, 5, 1],
    [0, 1, 0, 1],
    [0, 1, 0, 1]
]
`manhattan` =

17 LCS Backtrack

Funkcija ima dva parametra, niske `v` i `w`. Povratna vrednost je broj zajedničkih karaktera (broj pogodaka).

Prvo ćemo odrediti `n` - dužinu niske `v`, i `m` - dužinu niske `w`. Zatim, pravimo matricu cena `s`, dimenzije $(n+1) \times (m+1)$, i matricu `backtrack` istih dimenzija čije elemente inicijalizujemo na (-1, -1).

Za svaki element prve kolone, prethodnik je element iznad njega, a za svaki element prve vrste prethodnik je element levo od njega. Obe petlje započeti od 1 jer je element (0, 0) već inicijalizovan i ne upada u ove šablone!

Dvostrukom petljom obrađujemo ostale elemente. Za svakog računamo cenu kao maksimum tri vrednosti: cena gornjeg elementa (deletion), cena levog elementa (insertion) i cena dijagonalnog elementa uvećana za 1 ukoliko su karakteri na prethodnim pozicijama isti (ako su isti karakteri - match, ako su različiti - missmatch).

Potrebno je još odrediti prethodnika trenutnog elementa. To se može odrediti poređenjem cena trenutnog elementa sa cenama okolnih. Ukoliko je cena trenutnog jednaka ceni elementa iznad, onda je element iznad prethodnik. Ukoliko je cena trenutnog elementa jednaka ceni elementa levo od njega, onda je element levo njegov prethodnik. U suprotnom, element na dijagonali je prethodnik.

Preostalo je još da sastavimo nisko koja predstavlja zajedničku podsekvencu. Određujemo indekse, `i` i `j`, od kojih ćemo početi. To su indeksi prethodnika poslednjeg elementa. Niska `lcs` inicijalno je prazna. Ovu vrednost menjamo u slučaju da je prethodnik poslednjeg elementa na dijagonali, odnosno, u slučaju da je `i=n-1` i `j=m-1`. Tada, `lcs` dobija vrednost poslednjeg karaktera niske `v` (ili niske `w`, pošto su te vrednosti svakako jednake).

Petljuu vrtimo dok su `i` i `j` različiti od 0. U svakoj iteraciji proveravamo da li je prethodnih na dijagonali i, ako jeste, karakter prethodnika nadovezujemo na početak. Ažuriramo `i` i `j` tako da sadrže indekse prethodnika.

Kada je određena niska `lcs`, štampamo je i vraćamo cenu poslednjeg elementa.

```
1  def LCSBacktrack(v, w):
2    n = len(v)
3    m = len(w)
4    s = [[0 for j in range(m + 1)] for i in range(n + 1)]
5
6    backtrack = [[(-1,-1) for j in range(m + 1)] for i in range(n + 1)]
7
8    for i in range(1, n + 1):
9      backtrack[i][0] = (i-1, 0)
10
11   for j in range(1, m + 1):
12     backtrack[0][j] = (0, j-1)
13
14   for i in range(1, n + 1):
15     for j in range(1, m + 1):
16
17       s[i][j] = max(s[i-1][j], s[i][j-1], s[i-1][j-1] + int(v[i-1] == w[j-1]))
18
19       if s[i][j] == s[i-1][j]:
20         backtrack[i][j] = (i-1, j)
21       elif s[i][j] == s[i][j-1]:
22         backtrack[i][j] = (i, j-1)
23       else:
24         backtrack[i][j] = (i-1, j-1)
25
26   i = backtrack[n][m][0]
27   j = backtrack[n][m][1]
28
29   lcs = ""
30
31   if i == n-1 and j == m - 1:
32     lcs = v[n-1]
33
34   while not (i == 0 and j == 0):
35     if backtrack[i][j] == (i-1, j-1):
36       lcs = v[i-1] + lcs
37
38     i = backtrack[i][j][0]
39     j = backtrack[i][j][1]
40
41   print(lcs)
42
43   return s[n][m]
```

17.1 Test primer

v = „abcd”
w = „dabe”
lcs =
match =

18 Global Alignment

Funkcija ima dva parametra, niske v i w. Povratna vrednost je skor poravnanja.

Pre implementacije funkcije potrebno je definisati vrednosti za pogodak (MATCH), promašaj (MISSMATCH) i za rupu (GAP_PENALTY).

Prvo ćemo odrediti n - dužinu niske v, i m - dužinu niske w. Zatim, pravimo matricu cena s, dimenzije $(n+1) \times (m+1)$, i matricu backtrack istih dimenzija čije elemente inicijalizujemo na (-1, -1).

Za svaki element prve kolone, cenu računamo kao zbir cene elementa iznad njega i kazne za rupu. Prethodnik je element iznad njega jer smo njegovu cenu iskoristili. Za svaki element prve vrste, cenu računamo kao zbir cene elementa levo od njega i kazne za rupu. Prethodnik je element levo od njega jer smo njegovu cenu iskoristili. Obe petlje započeti od 1 jer je element (0, 0) već inicijalizovan i ne upada u ove šablone!

Dvostrukom petljom obrađujemo ostale elemente. Za svakog računamo cenu kao maksimum tri vrednosti: zbir cene gornjeg elementa i kazne za rupu (deletion), zbir cene levog elementa i kazne za rupu (insertion) i cena dijagonalnog elementa uvećana za za vrednost koju vraća funkcija match_score (18.1).

Potrebno je još odrediti prethodnika trenutnog elementa. To se može odrediti poređenjem cena trenutnog elementa sa cenama okolnih. Ukoliko je cena trenutnog jednaka zbiru cene elementa iznad i kazne za rupu, onda je element iznad prethodnik. Ukoliko je cena trenutnog elementa jednaka zbiru cene elementa levo od njega i kazne za rupu, onda je element levo njegov prethodnik. U suprotnom, element na dijagonali je prethodnik.

Sledeći korak je da odredimo kako izgledaju niske v i w sa svim insercijama i delecijama. Izmenjene niske čuvamo u promenljivama v_p i w_p, a njihovi indeksi su, redom, i=n i j=m. Niske su, naravno, inicijalno prazne.

Petlju vrtimo dok ne dođemo do početnog elementa, (0, 0). U zavisnosti od prethodnika zaključićemo da li se desila insercija, delecija ili (ne)poklapanje. Ukoliko je prethodnik dijagonalni element, u pitanju je (ne)poklapanje i obe niske proširujemo nadovezivanjem prethodnih karaktera na početak. Ukoliko je prethodnik element iznad, u pitanju je delecija. Niska v_p proširuje se nadovezivanjem prethodnog karaktera niske v na početak, a niski w_p se nadovezuje '-' na početak. U preostalom slučaju imamo inserciju kada se niski v_p nadovezuje '-' na početak, a niski w_p se nadovezuje prethodni karakter niske w na početak.

Nakon što smo formirali niske i odštampali ih na standardni izlaz, vraćamo ukupan skor ovog poravnanja, odnosno, vrednost poslednjeg elementa matrice s.

```
1  def global_alignment(v, w):
2    n = len(v)
3    m = len(w)
4
5    backtrack = [[(-1, -1) for j in range(m + 1)] for i in range(n + 1)]
6    s = [[0 for j in range(m + 1)] for i in range(n + 1)]
7
8    for i in range(1, n + 1):
9      s[i][0] = s[i-1][0] + GAP_PENALTY
10     backtrack[i][0] = (i-1, 0)
11
```

```
12    for j in range(1, m + 1):
13      s[0][j] = s[0][j-1] + GAP_PENALTY
14      backtrack[0][j] = (0, j-1)
15
16    for i in range(1, n + 1):
17      for j in range(1, m + 1):
18        s[i][j] = max(
19                s[i-1][j] + GAP_PENALTY,
20                s[i][j-1] + GAP_PENALTY,
21                s[i-1][j-1] + match_score(v[i-1], w[j-1])
22               )
23
24        if s[i][j] == s[i-1][j] + GAP_PENALTY:
25          backtrack[i][j] = (i-1, j)
26
27        elif s[i][j] == s[i][j-1] + GAP_PENALTY:
28          backtrack[i][j] = (i, j-1)
29        else:
30          backtrack[i][j] = (i-1, j-1)
31
32    v_p = ""
33    w_p = ""
34
35    i = n
36    j = m
37
38    while (i,j) != (0,0):
39      if backtrack[i][j] == (i-1, j-1):
40        v_p = v[i-1] + v_p
41        w_p = w[j-1] + w_p
42
43      elif backtrack[i][j] == (i-1, j):
44        v_p = v[i-1] + v_p
45        w_p = '-' + w_p
46
47      else:
48        v_p = '-' + v_p
49        w_p = w[j-1] + w_p
50
51      (i,j) = backtrack[i][j]
52
53    print(v_p)
54    print(w_p)
55
56    return s[n][m]
```

18.1 Match Score

Funkcija ima dva parametra, karaktere `c1` i `c2`. Funkcija vraća vrednost `MATCH`, ako su karakteri jednaki, odnosno, vraća `MISSMATCH` ako se razlikuju.

```
1  def match_score(c1, c2):
2    if c1 == c2:
```

```
    return MATCH
  else:
    return MISSMATCH
```

18.1.1 Test primer

v = ,,AAATTTGGGCCCGGGAAATTTCCC”
w = ,,GGGCCCTT”

19 Local Alignment

Funkcija ima dva parametra, niske `string_1` i `string_2`. Povratna vrednost je dužina najduže zajedničke podsekvence.

Pravimo matricu `DP`, dimenzije |string_1|×|string_2|, u kojoj čuvamo cene. Elementi prve kolone i prve vrste imaju vrednosti 0. Ostale elemente računamo u dvostrukoj petlji. Vrednost svakog elementa predstavlja maksimum tri četiri vrednosti: 0 (besplatna taksi vožnja), vrednost elementa iznad umanjena za 2 (delecija), vrednost elementa levo umanjena za 2 (insercija) i vrednost dijagonalnog elementa uvećana za jedan, ako su karakteri na prethodnoj poziciji jednaki, odnosno, nepormenjena ako se ti karakteri razlikuju.

Preostaje nam da odredimo maksimalnu vrednost matrice, jednostavnim proslaskom pomoću dvostruke petlje nakon čega ćemo tu vrednost vratiti.

```
def local_alignment(string_1, string_2):
  DP = [[0 for j in range(len(string_2) + 1)] for i in range(len(string_1) + 1)
    ↪ ]

  for i in range(len(string_1) + 1):
    DP[i][0] = 0

  for j in range(len(string_2) + 1):
    DP[0][j] = 0

  for i in range(1, len(string_1) + 1):
    for j in range(1, len(string_2) + 1):
      DP[i][j] = max(0, DP[i-1][j] - 2, DP[i][j-1] - 2, DP[i-1][j-1] + int(
    ↪ string_1[i-1] == string_2[j-1]))


  maximum = 0

  for i in range(len(DP)):
    for j in range(len(DP[i])):
      if DP[i][j] > maximum:
        maximum = DP[i][j]

  return maximum
```

19.1 Test primer

string_1 = ’ACGTGCTCG’
string_2 = ’AATGCTCT’

20 Edit Distance

Funkcija ima dva parametra, niske `v` i `w`. Povratna vrednost je edit rastojanje niski `v` i `w`, odnosno broj operacija potreban da bi se niska `v` pretvorila u nisku `w`.

Podsetimo se edit operacija: umetanje u prvu nisku (desno), brisanje iz druge niske (dole) i izmena karaktera prve niske u karakter druge niske (dijagonalno). Napomena: smer dijagonalno se koristi i kada se karakteri poklapaju, s tim da se tada edit rastojanje ne uvećava.

Prvo ćemo odrediti `n` - dužinu niske `v`, i `m` - dužinu niske `w`. Zatim, pravimo matricu rastojanja `s`, dimenzije $(n+1) \times (m+1)$, i matricu `backtrack` istih dimenzija čije elemente inicijalizujemo na (-1, -1).

Za svaki element prve kolone, rastojanje je redni broj kolone. Prethodnik je element iznad njega. Za svaki element prve vrste, rastojanje je redni broj vrste. Prethodnik je element levo od njega. Obe petlje započeti od 1 jer je element (0, 0) već inicijalizovan i ne upada u ove šablone!

Dvostrukom petljom određujemo vrednosti ostalih elemenata. Vrednost trenutnog elementa računa se kao minimum tri vrednosti: vrednost elementa iznad uvećana za 1 (delecija), vrednost elementa levo uvećana za 1 (insercija), vrednost dijagonalnog elementa, ako su karakteri na prethodnoj poziciji jednaki (poklapanje), inače je uvećana za jedan (nepoklapanje).

Potrebno je još odrediti prethodnika trenutnog elementa. To se može odrediti poređenjem rastojanja trenutnog elementa sa rastojanjima okolnih. Ukoliko je rastojanje trenutnog jednako zbiru rastojanja elementa iznad uvećanog za 1, onda je element iznad prethodnik. Ukoliko je rastojanje trenutnog elementa jednako zbiru rastojanja elementa levo od njega uvećanog za 1, onda je element levo njegov prethodnik. U suprotnom, element na dijagonali je prethodnik.

Sledeći korak je da odredimo kako izgledaju niske `v` i `w` sa svim insercijama i delecijama. Izmenjene niske čuvamo u promenljivama `v_p` i `w_p`, a njihovi indeksi su, redom, `i=n` i `j=m`. Niske su, naravno, inicijalno prazne.

Petlju vrtimo dok ne dođemo do početnog elementa, (0, 0). U zavisnosti od prethodnika zaključićemo da li se desila insercija, delecija ili (ne)poklapanje. Ukoliko je prethodnik dijagonalni element, u pitanju je (ne)poklapanje i obe niske proširujemo nadovezivanjem prethodnih karaktera na početak. Ukoliko je prethodnik element iznad, u pitanju je delecija. Niska `v_p` proširuje se nadovezivanjem prethodnog karaktera niske `v` na početak, a niski `w_p` se nadovezuje '-' na početak. U preostalom slučaju imamo inserciju kada se niski `v_p` nadovezuje '-' na početak, a niski `w_p` se nadovezuje prethodni karakter niske `w` na početak.

Nakon što smo formirali niske i odštampali ih na standardni izlaz, vraćamo ukupan skor ovog poravnanja, odnosno, vrednost poslednjeg elementa matrice `s`.

```
1  def edit_distance(v, w):
2    n = len(v)
3    m = len(w)
4
5    s = [[0 for j in range(m + 1)] for i in range(n + 1)]
6    backtrack = [[(-1, -1) for j in range(m + 1)] for i in range(n + 1)]
7
8    for i in range(1, n + 1):
9      s[i][0] = i
10     backtrack[i][0] = (i-1, 0)
11
12   for j in range(1, m + 1):
13     s[0][j] = j
14     backtrack[0][j] = (0, j-1)
15
16   for i in range(1, n + 1):
17     for j in range(1, m + 1):
18       s[i][j] = min(s[i-1][j] + 1, s[i][j-1] + 1, s[i-1][j-1] + int(v[i-1] != w
```

```
    ↪ [j-1]))
19
20        if s[i][j] == s[i-1][j] + 1:
21          backtrack[i][j] = (i-1, j)
22
23        elif s[i][j] == s[i][j-1] + 1:
24          backtrack[i][j] = (i, j-1)
25        else:
26          backtrack[i][j] = (i-1, j-1)
27
28    v_p = ""
29    w_p = ""
30
31    i = n
32    j = m
33
34    while (i,j) != (0,0):
35      if backtrack[i][j] == (i-1, j-1):
36        v_p = v[i-1] + v_p
37        w_p = w[j-1] + w_p
38
39      elif backtrack[i][j] == (i-1, j):
40        v_p = v[i-1] + v_p
41        w_p = '-' + w_p
42
43      else:
44        v_p = '-' + v_p
45        w_p = w[j-1] + w_p
46
47      (i,j) = backtrack[i][j]
48
49    print(v_p)
50    print(w_p)
51
52    return s[n][m]
```

20.1 Test primer

v = „AAATTTGGGCCCGGGAAATTTCCC”
w = „AAACCCTTTGGGCCCTTTAAACCC’

21 Affine Gap Alignment

Funkcija ima dva aprametra, niske `string_1` i `string_2`. Povratna vrednost je vrednost najvećeg poravnanja.

Održavamo tri matrice, `upper`, `lower` i `middle`. Svaka je dimenzija (|string_1|+1) × (|string_2|+1) i svakoj su vrednosti inicijalizovane na $-\infty$. Jedino prvi element srednje matrice ima vrednost 0.

U dvostrukoj petlji računamo vrednosti za svaki element ovih matrica kao maksimume tri broja. Prelazak sa jednog nivoa na drugi, odnosno, iz jedne matrice u drugu, kažnjavamo sa -10 (to je pojava nove rupe - gap). Svaki sledeći karakter u rupi kažnjavamo sa -0.5.

Za gornju matricu to su: element središnje matrice iz prethodne kolone umanjen za 10 i 0.5, element gornje matrice iz prethodne kolone umanjen za 0.5 i element donje matrice umanjen za 10 i 0.5.

Za donju matricu to su: element središnje matrice iz prethodne vrste umanjen za 10 i 0.5, element gornje matrice iz prethodne vrste umanjen za 10 i 0.5 i element donje matrice umanjen za 0.5.

Za središnju matricu to su: element središnje matrice iz prethodne kolone i prethodne vrste uvećan za povratnu vrednost funkcija `match` za karaktere niski na prethodnim pozicijama (21.1), element gornje matrice i element donje matrice.

Povratna vrednost je maksimum poslednjih elemenata ove tri matrice.

```
1  def affine_gap_alignment(string_1, string_2):
2    upper = [[float('-inf') for j in range(len(string_2) + 1)] for i in range(len
       ↪ (string_1) + 1)]
3    lower = [[float('-inf') for j in range(len(string_2) + 1)] for i in range(len
       ↪ (string_1) + 1)]
4    middle = [[float('-inf') for j in range(len(string_2) + 1)] for i in range(
       ↪ len(string_1) + 1)]
5
6    middle[0][0] = 0
7
8    for i in range(1, len(string_1) + 1):
9      for j in range(1, len(string_2) + 1):
10       upper[i][j] = max(-10 -0.5 + middle[i][j-1], (-0.5 + upper[i][j-1]), (-10
       ↪  -0.5 + lower[i][j-1]))
11       lower[i][j] = max(-10 -0.5 + middle[i-1][j], (-10 -0.5 + upper[i-1][j]),
       ↪ (-0.5 + lower[i-1][j]))
12       middle[i][j] = max(match(string_1[i-1], string_2[j-1]) + middle[i-1][j
       ↪ -1], upper[i][j], lower[i][j])
13
14   return max(upper[len(string_1)][len(string_2)], middle[len(string_1)][len(
       ↪ string_2)], lower[len(string_1)][len(string_2)])
```

21.1 Match

Funkcija ima dva parametra, karakteri `s1` i `s2`. Ukoliko su karakteri jednaki, vraćamo vrednost 1, u suprotnom -4.

```
1  def match(s1, s2):
2    if s1 == s2:
3      return 1
4    else:
5      return -4
```

21.1.1 Test primer

string_1 = 'ACGTGCTCG'
string_2 = 'AATGCTCT'
alignment =

22 Greedy Sorting

Funkcija ima jedan parametra, permutaciju blokova `P` koje treba sortirati rastuće tako da svi budu pozitivne orijentacije. Povratna vrednost je broj obrtanja koji je bio potreban da bi se došlo do odgovarajućeg poretka. Pored toga, funkcija ispisuje izgled permutacije na kraju svake iteracije.

Ideja je da redom svaki blok smeštamo na njegovo mesto. Prvo 1. blok treba smestiti na prvo mesto i obezbediti da bude pozitivno orijentisan. Zatim, smeštamo drugi blok, pa treći i tako redom dok ne smestimo sve na odgovarajuća mesta.

Prvo ispisujemo početnu permutaciju. Indeks `k` predstavlja broj bloka koji u toj iteraciji treba smestiti na odgovarajuće mesto. Prvo proverimo da li se blok možda nalazi na svom mestu (obratiti pažnju da indeksiranje ide od 0, a ne od 1, tako da je mesto za prvi blok 0, a ne 1). Ako jeste, nemamo šta da radimo pa prelazimo na sledeću iteraciju. Ako nije, određujemo indeks na kom se nalazi taj blok (funkcijom `find`, 22.1). Zatim, vršimo obrtanje funkciojom `reversal` (22.2) i uvećavamo ukupan broj obrtanja.

Ostalo je još da proverimo da li je blok ispravno orijentisan. Ako je potizitivan, onda je u redu. Međutim, ako nije, treba da ga obrnemo. TO činimo jednostavnim negiranjem trenutne vrednosti i još jednom uvećavamo ukupan broj rotacija za jedan. Ispisujemo trenutnu permutaciju i prelazimo na sledeću iteraciju.

```
1  def greedy_sorting(P):
2    approx_reversal_distance = 0
3
4    print(P)
5
6    for k in range(len(P)):
7      if P[k] != k+1:
8        i = find(P, k, k+1)
9        P = reversal(P, k, i)
10       approx_reversal_distance += 1
11
12       print(P)
13
14       if P[k] < 0:
15         P[k] = -P[k]
16         approx_reversal_distance += 1
17
18         print(P)
19
20   return approx_reversal_distance
```

22.1 Find

Funkcija ima tri parametra, permutaciju `P`, `start` - početnu poziciju dela koji obrćemo, i redni broj bloka koji tražimo `n`. Povratna vrednost je indeks na kom se nalazi taj blok.

Petlju počinjemo od `start` i idemo do kraja permutacije. Pošto blokovi mogu biti označeni i negativnim brojem, bitno je da proverimo da li je trenutni element koji ispitujemo jednak `n` ili `-n`. Kada naiđemo da takav element, vraćamo njegov indeks.

```
1  def find(P, start, n):
2    for i in range(start, len(P)):
3      if P[i] == n or P[i] == -n:
4        return i
```

22.2 Reversal

Funkcija ima tri parametra, permutaciju `P`, `start` - početnu poziciju dela koji obrćemo, i `stop` - krajnju poziciju dela koji obrćemo. Funkcija vraća izmenjenu permutaciju.

Podsetimo se da rotacija obrće blokove iz intervala pri čemu im menja i orijentaciju (odnosno, znak). U promenljivu `rev` upisaćemo negativne vrednosti elemenata permutacije od pozicije `start` do pozicije `stop` (uključujući i tu poziciju). Zatim, obrćemo listu `rev`. Ostaje još da u `P` upišemo `rev` od pozicije `start` do pozicije `stop`, nakon čega vraćamo `P`.

```
1  def reversal(P, start, stop):
2    rev = [-i for i in P[start:stop+1]]
3    rev.reverse()
4
5    P[start:stop+1] = rev
6
7    return P
```

22.2.1 Test primer 1

`P` = [+1,-7,+6,-10,+9,-8,+2,-11,-3,+5,+4]
`approx_reversal_distance` =

22.2.2 Test primer 1

`P` = [-2,-5,3,4,1]
`approx_reversal_distance` =

23 Chromosome To Cycle

Funkcija ima jedan parametar, hromozom `chromosome` koji pretvaramo u ciklus. Povratna vrednost je lista čvorova ciklusa.

Polazimo od liste nula dimenzije 2·|`chromosome`|. Obrađujemo jedan po jedan element hromozoma i od svakog dobijamo dva čvora. Ukoliko je element pozitivan, na parnu poziciju smeštamo njegovu dvostruku vrednost umanjenu za 1, a na neparnu poziciju smeštamo njegovu dvostruku vrednost. U suprotnom, ako je element negativan, na parnu poziciju smeštamo negiranu njegovu dvostruku vrednost, a na neparnu poziciju smeštamo njegovu negiranu dvostruku vrednost umanjenu za 1.

```
1  def chromosome_to_cycle(chromosome):
2
3    nodes = [0 for i in range(2*len(chromosome))]
4
5    for j in range(len(chromosome)):
6      i = chromosome[j]
7      if i > 0:
8        nodes[2*j] = 2*i -1
9        nodes[2*j+1] = 2*i
10     else:
11       nodes[2*j] = -2*i
12       nodes[2*j+1] = -2*i-1
13   return nodes
```

23.1 Test primer

`P` = [1,-2,-3,4]
`nodes` =

24 Cycle To Chromosome

Funkcija ima jedan parametar, listu čvorova `nodes` ciklusa od kog treba sastaviti ciklus. Povratna vrednost je hromozom.

Pošto svakom bloku hromozoma odgovaraju dva čvora, ovde ćemo koristiti samo pola niza. Lista blokova `chromosomes` je upola manje dimenzije od liste čvorova i inicijalno sadrži smao vrednosti 0.

Indeks u petlji takođe ide do pola dužine liste `nodes`. Ukoliko je čvor na parnoj poziciji manji od onog na neparnoj, onda je odgovarajući blok pozitivan i njegovu vrednost ćemo dobiti polovljenjem vrednosti neparnog čvora. U suprotnom, blok je negativan i njegova vrednost dobija se polovljenjem vrednosti negativnog parnog čvora.

```
1  def cycle_to_chromosome(nodes):
2
3    chromosomes = [0 for i in range(len(nodes)//2)]
4
5    for j in range(len(nodes)//2):
6      if nodes[2*j] < nodes[2*j+1]:
7        chromosomes[j] = nodes[2*j+1] // 2
8      else:
9        chromosomes[j] = -nodes[2*j] // 2
10
11   return chromosomes
12
```

24.1 Test primer

`nodes` = [1, 2, 4, 3, 6, 5, 7, 8]
`chromosome` =

25 Shortest Rearrangement Scenario

Funkcija ima dva parametra, hromozome `P` i `Q`. Povratna vrednost je broj izmena potreban da se od jedne permutacije dođe do druge.

Određujemo crvene i plave grane (funkcijom `colored_edges`, 25.1). Petlja se izvršava dok ima trivijalnih ciklusa (što proveravamo funkcijom `has_nontrivial_cycle`, 25.2). Biramo jednu granu iz tog ciklusa (funkcijom `select_edge_from_nontrivial_cycle`, 25.3). Označićemo sa `j` izlazni čvor grane, a sa `i_p` ulazni čvor te grane. Pokušaćemo još da odredimo vrednosti za čvorove `i` i `j_p` takve da su `i` i `j` povezani u `P`, i `i_p` i `j_p` povezani u `P`, a inicijalizujemo ih sa -1.

Prolazimo sve crvene grane, i ako naiđemo na čvor `j` među ulaznim ili izlaznim čvorovima, onda će `i` dobiti vrednost suprotnog čvora. Takođe, ako naiđemo na `i_p` među izlaznim čvorovima, onda će `j_p` dobiti vrednost odgovarajućeg ulaznog čvora.

Dakle pronašli smo čvorove takve da su povezani `i` i `j`, `i_p` i `j_p`, a iz `Q` smo izabrali granu koja povezuje `j` i `i_p` koji. Iz crvenih grana uklanjamo grane koje povezuju `i` i `j`, `i_p` i `j_p` (ima ih po dve, po jedna za svaki pravac). Dodajemo grane koje će povezivati `j` sa `i_p`, i `j_p` sa `i` (opet, po dve za oba slučaja). U ovom trenutku povećavamo broj prekida.

```
def shortest_rearrangement_scenario(P, Q):
  red_edges = colored_edges([P])
  blue_edges = colored_edges([Q])

  num_of_breaks = 0

  while has_nontrivial_cycle(red_edges, blue_edges):
    (j,i_p) = select_edge_from_nontrivial_cycle(red_edges, blue_edges)

    i = -1
    j_p = -1

    for (v,w) in red_edges:
      if v == j:
        i = w
      if w == j:
        i = v
      if v == i_p:
        j_p = w

    red_edges.remove((j, i))
    red_edges.remove((i, j))

    red_edges.remove((i_p, j_p))
    red_edges.remove((j_p, i_p))

    red_edges.append((j, i_p))
    red_edges.append((i_p, j))

    red_edges.append((j_p, i))
    red_edges.append((i, j_p))

    num_of_breaks += 1
```

25.1 Colored Edges

Funkcija ima jedan parametar, permutaciju `P`. Povratna vrednost je lista obojenih grana.

Krećemo od prazne liste. Prolazimo redom sve hromozome u `P`. Svaki ćemo prvo pretvoriti u ciklus. Zatim prolazimo čvorove (do pola dužine, ili do dužine hromozoma) i u listu grana dodajemo dve grane, pošto su obojene grane neusmerene, a naš graf je usmeren. Grane sadrži neparan čvor i paran čvor po modulu broja čvorova.

```
def colored_edges(P):
    edges = []
    for chromosome in P:
        nodes = chromosome_to_cycle(chromosome)
        for j in range(len(chromosome)):
            edges.append((nodes[2*j+1], nodes[(2*j+2)
    len(nodes)]))edges.append((nodes[(2*j+2)  len(nodes)], nodes[2*j+1]))

    return edges
```

25.2 Has Nontrivial Cycle

Funkcija ima dva parametra, `P` i `Q`. Povratna vrednost je boolean tipa.

Prolazimo sve grane u `P`. Ukoliko postoji braj jedna grana takva da ne postoji ni ta grana ni njena inverzija u `Q`, vratiti `True`. U suprotnom, ako su sve grane iz `P` prisutne u `Q`, vraćamo `False`.

```
1  def has_nontrivial_cycle(P, Q):
2    for (v,w) in P:
3      if (v,w) not in Q and (w,v) not in Q:
4        return True
5
6    return False
```

25.3 Select Edge From Nontrivial Cycle

Funkcija ima dva parametra, `P` i `Q`. Povratna vrednost je grana iz `Q` koja se ne nalazi u `P`.

Petljom prolazimo sve grane iz `Q`. Vraćamo prvu granu koja se ne nalazi u `P` niti u istom obliku niti u suprotnom.

```
1  def select_edge_from_nontrivial_cycle(P, Q):
2    for (v,w) in Q:
3      if (v,w) not in P and (w,v) not in P:
4        return (v,w)
```

25.3.1 Test primer

`P` = [1,-2,-3,4]
`Q` = [1, 2, 3, -4]
`num_of_breaks` =

26 Two Break On Genome

Funkcija ima pet parametara, `P` i čvorove `i`, `i_p`, `j` i `j_p`. Povratna vrednost je graf dobijen genoma `P`.

Prvo određujemo crne i crvene grane nadovezivanjem povratnih vrednosti funkcija `black_edges` (26.1) i `colored_edges` (25.1). Zatim prespajamo čvorove funkcijom `two_break_on_genome_graph` (26.2). Na kraju pravimo graf funkcijom `graph_to_genome` (26.3).

```
1  def two_break_on_genome(P, i, ip, j, jp):
2    genome_graph = black_edges(P) + colored_edges([P])
3    genome_graph = two_break_on_genome_graph(genome_graph, i, ip, j, jp)
4
5    P = graph_to_genome(genome_graph)
6
7    return P
```

26.1 Black Edges

Funkcija ima jedan parametra, `P`. Povratna vrednost je lista crnih grana.

Polazimo od prazne liste grana i indeksa 0. Petlja se izvršava dok nismo obradili sve čvorove. Ukoliko je vrednost trenutnog čvora manja od vrednosti sledećeg, crna grana je usmerena od trenutnog ka narednom, u suprotnom je usmerena obrnuto. Indeks uvećavamo sa korakom 2.

```
def black_edges(P):
  nodes = chromosome_to_cycle(P)

  edges = []

  i = 0;

  while i < len(nodes):
    if nodes[i] < nodes[i+1]:
      edges.append((nodes[i], nodes[i+1]))
    else:
      edges.append((nodes[i+1], nodes[i]))

    i = i + 2

  return edges
```

26.2 Two Break On Genome Graph

Funkcija ima 5 parametara, listu grana `genome_graph` i čvorove `i`, `i_p`, `j` i `j_p` koje treba prespojiti. Povratna vrednost je lista novih grana.

Polazimo od nove, prazne, liste grana. Prolazimo sve grane iz liste i sve ih dodajemo u novu listu, osim grana `(i, i_p)` i `(j, j_p)`. Kada su obrađene sve grane, dodajemo još dve , `(i, j)` i `(i_p, j_p)`.

```
def two_break_on_genome_graph(genome_graph, i, ip, j, jp):

  new_edges = []

  for edge in genome_graph:
    if (edge[0] == i and edge[1] == ip) or (edge[0] == j and edge[1] == jp):
      continue
    new_edges.append(edge)

  new_edges.append((i,j))
  new_edges.append((ip,jp))

  return new_edges
```

26.3 Graph To Genome

Funkcija ima jedan parametar, listu grana `genome_graph`. Povratna vrednost je genom napravljen od dobijenog grafa.

Polazimo od praznog genoma `P` i prazne liste čvorova `nodes`. Prvo treba odrediti sve čvorove ovog grafa. To činimo prolaskom kroz listu grana i dodavanjem ova njena kraja u listu čvorova.

Zatim, poslednji čvor iz liste treba premestiti na početak. Pravimo dve liste. Prva sadrži samo poslednji čvor, druga sadrži ostatak. U listu `nodes` smeštamo rezultat nadovezivanja druge liste na prvu.

Sada pravimo hromozom na osnovu ove liste čvorova (funkcijom `cycle_to_chromosome`, 24). Povratnu vrednost funkcije nadovezujemo na `P` i to vraćamo iz funkcije.

```
def graph_to_genome(genome_graph):
  P = []
  nodes = []

  for (i,j) in genome_graph:
    nodes.append(i)
    nodes.append(j)


  prvi = [nodes[-1]]
  ostatak = copy.copy(nodes[:-1])
  nodes = prvi + ostatak

  chromosome = cycle_to_chromosome(nodes)
  P.append(chromosome)

  return P
```

27 Additive Phylogeny

Funkcija ima dva parametra, matricu `D` i broj čvorova `n`. Povratna vrednost je graf `T`.

Implementacija je rekurzivna. Izlaz iz rekurzije je slučaj kada imamo samo dva čvora. Tada pravimo graf sa ta dva čvora, a grane su otežane odgovarajućim vrednostima iz matrice `D`. Graf je predstavljen kao mapa, gde je ključ čvor, a vrednost je lista parova (čvor, težina grane).

Određujemo vrednost limb-a (funkcijom `limb`, 27.1). Svakoj grani (tj. svakoj vrednosti u matrici `D`), ne uključujući poslednji, umanjujemo vrednost za limb.

Određujemo tri lista takva da zbir težina dve grane bude jednak težini treće grane (funkcijom `three_leaves`, 27.2). Zapamtićemo vrednost grane `(i, n-1)` u promenljivu `x`.

Sada nam je potreban graf `T` za istu matricu i smanjen broj čvorova za 1 (rekurzivni poziv). Sada treba da ubacimo novi čvor u T. To ćemo učiniti funkcijom `insert_vertex_on_path` pri čemu nam ona vraća novi graf `T` i čvor `v`, koji je umetnut. Tom čvoru u listu dodajemo par `(n-1, limb)`, a čvoru `n-1` pravimo listu koja sadrži samo par `(v, limb)`.

```
1  def additive_phylogeny(D, n):
2    if n == 2:
3      return {
4        0: [(1, D[0][1])],
5        1: [(0, D[1][0])]
6      }
7
8    limb_length = limb(D, n-1)
9
10   for j in range(n-1):
11     D[j][n-1] = D[j][n-1] - limb_length
12     D[n-1][j] = D[j][n-1]
13
14   (i, n, k) = three_leaves(D, n)
15
16   x = D[i][n-1]
17
18   T = additive_phylogeny(D, n - 1)
19
20   (T, v) = insert_vertex_on_path(T, i, k, x)
21
22   T[v].append((n-1, limb_length))
23   T[n-1] = [(v, limb_length)]
24
25   return T
26
```

27.1 Limb

Funkcija ima dva parametra, matricu `D` i čvor `j`. Funkcija vraća najmanju vrednost rastojanja čvorova.

Dvostrukom petljom prolazimo sve čvorove i tražimo dva čvora `i` i `k` koji daju najmanje rastojanje po formuli (D[i][j] + D[j][k] - D[i][k])/2.

```
1  def limb(D, j):
2
3    minimum = float('inf')
4
```

```
  for i in range(j):
    for k in range(i+1, j):
      dist = (D[i][j] + D[j][k] - D[i][k])/2
      if dist < minimum:
        minimum = dist

  return int(minimum)
```

27.2 Three Leaves

Funkcija ima dva parametra, matricu `D` i čvor `n`. Funkcija vraća torku sačinjenu od tri čvora takva da važi da je težina grane `(i, k)` jednaka zbiru težina grana `(i, n-1)` i `(n-1, k)`.

```
def three_leaves(D, n):
  for i in range(n-1):
    for k in range(i, n-1):
      if D[i][k] == (D[i][n-1] + D[n-1][k]):
        return (i, n, k)
```

27.3 Insert Vertex On Path

Funkcija ima četiri parametra, trenutni graf `T`, čvorove `start` i `stop` i rastojanje između tih čvorova `distance`. Povratna vrednost je par novi graf `T` i čvor `new_v` koji je ubačen u graf.

Prvo moramo odrediti putanju u grafu `T` od čvora `start` do `stop` (funkcijom `find_path`, 27.3.1). Zatim, određujemo mesto gde treba ubaciti novi čvor (funkcijom `find_insertion_point`, 27.3.2). Funkcija vraća dva čvora `v` i `w`, između kojih ubacujemo čvor, rastojanje `e` između tih čvorova i rastojanje `e_dist` na kom treba da se nađe novi čvor. Ukoliko je vraćena vrednost 0 za rastojanje, vratiti par neizmenjeni graf `T` i čvor `w`.

Iz liste čvora `v` uklanjamo granu koja vodi do `w` i obrnuto. Novi čvor dobijamo nadovezivanjem `v`, `w` i `e_dist`. Njegovu listu postavljamo na praznu. Zatim, u nju nadovežemo granu do čvora `w` sa rastojanjem `e_dist`, i do čvora `v` sa rastojanjem `e - e_dist`. Takođe, treba dodati grane do novog čvora u liste za čvorove `v` i `w` sa odgovarajućim rastojanjima.

```
def insert_vertex_on_path(T, start, stop, distance):
  path = find_path(T, start, stop)
  (v, w, e, e_dist) = find_insertion_point(path, distance)

  if e_dist == 0:
    return (T, w)

  T[v].remove((w, e))
  T[w].remove((v, e))

  new_v = str(v) + str(w) + str(e_dist)

  T[new_v] = []

  T[new_v].append((w, e_dist))
  T[w].append((new_v, e_dist))

  T[new_v].append((v, e - e_dist))
  T[v].append((new_v, e - e_dist))
```

```
20
21    return (T, new_v)
```

27.3.1 Find Path

Funkcija ima tri parametra, trenutni graf `T` i čvorove `start` i `stop`. Povratna vrednost je lista grana, odnosno, parova (čvor, rastojanje).

Inicijalizujemo putanju na par početni čvor i rastojanje 0. Održavamo mapu posećenih čvorova i označavamo da je `start` posećen.

Dok u putanji ima parova, izdvajamo čvor `v` poslednje grane. Ukoliko je taj čvor jednak čvoru `stop`, vraćamo putanju. Inače, pokušavamo da pronađemo čvor `w` koji nije posećen, a povezan je sa `v`. Dakle, prolazimo listu čvora `v` i prvi par `(w, e)`, za koji važi da `w` nije posećen, dodajemo u putanju. Obeležavamo da je `w` posećen i da smo pronašli pogodan čvor. Pretragu prekinuti čim se nađe čvor.

Ukoliko nismo uspeli da pronađemo takav čvor, izbacujemo poslednji par iz putanje i nastavljamo dalje. Ako smo izbacili sve parove iz putanje, vraćamo praznu listu.

```
1  def find_path(T, start, stop):
2    path = [(start, 0)]
3    visited = {}
4    visited[start] = True
5
6    while len(path) > 0:
7      v = path[-1][0]
8
9      if v == stop:
10       return path
11
12     found = False
13
14     for (w,e) in T[v]:
15       if w not in visited:
16         path.append((w,e))
17         visited[w] = True
18         found = True
19         break
20
21     if not found:
22       path.pop()
23
24   return []
```

27.3.2 Find Insertion Point

Funkcija ima dva parametra, putanju `path` i rastojanje `distance`. Povratna vrednost torka koja sadrži dva čvora, između kojih treba ubaciti novi čvor, rastojanje između ta dva čvora i rastojanje na kom novi čvor treba da se nađe u odnosu na drugi čvor.

Pamtimo trenutno rastojanje, koje je incijijalno jednako 0, i prethodni čvor, koji je na početku prvi čvor na putanji.

Za svaki sledeći element u `path` izdvajamo čvor `v` i rastojanje `e` iz tog para. Trenutno rastojanje uvećavamo za `e` i pamtimo `v` u `next_vertex`. Ukoliko je trenutno rastojanje veće od `distance` našli smo čvorove koji su nam potrebni. Vraćamo torku (prethodni čvor, sledeći čvor,

`e`, razlika trenutnog i zadatog rastojanja). Inače, prehodni čvor postavljamo na `v` i nastavljamo potragu.

```
def find_insertion_point(path, distance):
  current_distance = 0
  previous_vertex = path[0][0]

  for i in range(1, len(path)):
    v = path[i][0]
    e = path[i][1]

    current_distance += e
    next_vertex = v


    if distance <= current_distance:
      return (previous_vertex, next_vertex, e, current_distance - distance)

    previous_vertex = v
```

27.3.3 Test primer

```
D = [
    [0, 13, 21, 22],
    [13, 0, 12, 13],
    [21, 12, 0, 13],
    [22, 13, 13, 0]
]
n = len(D)
T =
```

28 UPGMA

Funkcija ima dva parametra, matricu rastojanja `D` i broj čvorova `n`. Povratna vrednost je prvi klaster.

Inicijalizujemo listu klastera tako da svaki klaster sadrži jedan element - `i`, i svaki je starosti 0. Broj klastera, `num_of_clusters` postavljamo na broj redova matrice `D`.

Dok je broj klastera veći od 1, tražimo dva najsličnija klastera (rastojanje se koristi kao mera sličnosti i koristimo funkciju `min_distance`, 28.2). Funkcija nam vraća `i` i `j` - indekse bliskih klastera, kao i `distance` - njihovo rastojanje.

Pravimo novi klaster koji sadrži elemente oba klastera, a njegova starost je polovina rastojanja. Levo dodajemo `i`-ti, a desno `j`-ti klaster.

Ostaje još da ažuriramo listu klastera. U novu listu nadovezujemo sve klastere osim `i`-tog i `j`-tog. Zatim, dodajemo novi klaster. Prepisujemo novu listu u početnu listu klastera, a broj klastera smanjujemo za 1.

```
def UPGMA(D, n):
  clusters = [Cluster([i], 0) for i in range(n)]

  num_clusters = len(D)

  while num_clusters > 1:
    (i, j, distance) = min_distance(clusters, D, num_clusters)
```

```
8
9       new_cluster = Cluster(clusters[i].elements + clusters[j].elements, distance
        ↪ /2)
10      new_cluster.add_left(clusters[i])
11      new_cluster.add_right(clusters[j])
12
13      new_clusters_list = []
14      for c in range(num_clusters):
15        if c != i and c != j:
16          new_clusters_list.append(clusters[c])
17
18      new_clusters_list.append(new_cluster)
19      clusters = new_clusters_list[:]
20      num_clusters -= 1
21
22    return clusters[0]
```

28.1 Cluster

Klasa sadrži polje `elements`, `age`, `left` i `right`. Prva dva se inicijalizuju vrednostima parametara, a ostali imaju početnu vrednost `None`.

Funkcija za ispit, `str`, prvo ispisuje elemente klastera a potom i starost.

Funkcija `add_left` (`add_right`) postavlja vrednost polja `left` (`right`) na vrednost parametra.

```
1  class Cluster:
2    def __init__(self, elements, age):
3      self.elements = elements
4      self.age = age
5      self.left = None
6      self.right = None
7
8    def __str__(self):
9      return ("%s : %d" % (self.elements, self.age))
10
11   def add_left(self, L):
12     self.left = L
13
14   def add_right(self, R):
15     self.right = R
```

28.2 Min distance

Funkcija ima tri parametra, listu klastera `clusters`, matricu rastojanja `D` i broj klastera `num_of_clusters`. Povratna vrednost je torka - dva redna broja klastera koji su najbliži i njihovo rastojanje.

Pamtimo trenutno najmanje rastojanje, i indekse klastera koji se nalaze natom najmanjem rastojanju. Dvostrukom petljom prolazimo sve klastere iz liste. Za svaki par odredićemo rastojanje (funkcijom `distance`, 28.2.1). Ukoliko je to rastojanje manje od trenutnog minimuma, ažuriramo minimum i indekse. Na kraju vraćamo torku koja sadrži te tri vrednosti.

```
1  def min_distance(clusters, D, num_clusters):
```

```
2
3   minimum = float('inf')
4   min_i = -1
5   min_j = -1
6
7   for i in range(num_clusters):
8     for j in range(i+1, num_clusters):
9       dist = distance(D, clusters[i], clusters[j])
10      if dist < minimum:
11        minimum = dist
12        min_i = i
13        min_j = j
14
15  return (min_i, min_j, minimum)
```

28.2.1 Distance

Funkcija ima tri parametra, matricu rastojanja `D` i dva klastera `cluster_1` i `cluster_2`. Povratna vrednost je rastojanje ta dva klastera.

Rastojanje se računa kao količnik sume rastojanja svih elemenata ova dva klastera i proizvoda njihovog broja elemenata.

```
1   def distance(D, cluster_1, cluster_2):
2     d = 0
3     n1 = len(cluster_1.elements)
4     n2 = len(cluster_2.elements)
5
6     for i in cluster_1.elements:
7       for j in cluster_2.elements:
8         d += D[i][j]
9
10    return d/(n1*n2)
```

28.2.2 Test primer

`D` = [
    [0, 13, 21, 22],
    [13, 0, 12, 13],
    [21, 12, 0, 13],
    [22, 13, 13, 0]
]
`n` = len(D)
`T` =

29 Trie Matching

Funkcija ima dva parametra, `text` i sekvence koje tražimo u tekstu u drvenolikoj strukturi - `Trie`. Funkcija vraća listu sekvenci koje se nalaze u tekstu.

Krećemo od prazne liste pronađenih sekvenci. Petlju vrtimo dok nismo izbacili sve karaktere iz teksta. Tražimo sekvence koje se nalaze u trenutnom tekstu (funkcijom `preficTriePatternMatching`, 29.1). Ukoliko smo pronašli neku, dodajemo povratnu vrednost u listu. Uklanjamo prvi karakter iz teksta i ponavljamo postupak.

```
1  def trie_matching(text, Trie):
2    found_patterns = []
3    while len(text) > 0:
4      res = prefix_trie_pattern_matching(text, Trie)
5      if res != False:
6        found_patterns.append(res)
7      text = text[1:]
8    return found_patterns
```

29.1 Prefix Trie Pattern Matching

Funkcija ima dva parametra, `text` i sekvence koje tražimo u tekstu u drvenolikoj strukturi - `Trie`. Funkcija vraća sekvencu iz `Trie` koja se nalazi u tekstu.

Čvor `v` na početku je koreni i dodeljujemo mu vrednost 'root'. Prolazimo redom sve karaktere teksta. Ukoliko se neki od karaktera ne nalazi u listi čvora `v` vraćamo `False`. Inače, `v` dobija novu vrednost, i to onu koja odgovara karakteru `c` u listi čvora `v`.

Sada proveravamo da li se znak '$' nalazi u listi čvora `v`. Ako se nalazi, vraćamo vrednost koja odgovara karakteru '$' u listi čvora `v`. Ako smo obradili sve karaktere teksta, znači da nismo naišli na poklapanje i vraćamo `False`.

```
1  def prefix_trie_pattern_matching(text, Trie):
2    v = 'root'
3
4    for c in text:
5      if c not in Trie[v]:
6        return False
7
8      v = Trie[v][c]
9
10     if '$' in Trie[v]:
11       return Trie[v]['$']
12
13   return False
```

29.2 Trie Construction

Funkcija ima jedan parametar, listu `patterns` od koje treba napraviti `Trie`, što je povratna vrednost.

`Trie` će biti predstavljeno mapom čiji su ključevi nazivi čvorova (neke niske), a vrednosti su mape.

Inicijalizujemo `Trie` na praznu mapu, a onda dodajemo čvor sa nazivom *'root'* čija je mapa takođe prazna na početku. Dodatno, pratimo broj čvorova, na početku to je 1.

Prolazimo redom sve sekvence liste `patterns` i dodajemo ih u `Trie` (funkcijom `add_to_trie`, 29.3). Obratiti pažnju da treba proslediti sekvencu koja sadrži znak '$' na kraju! Funkcija vraća izmenjeni `Trie` i broj čvorova.

```
def trie_construction(patterns):
  Trie = {}
  Trie['root'] = {}

  number_of_nodes = 1

  for i in range(len(patterns)):
    pattern = patterns[i]
    (Trie, number_of_nodes) = add_to_trie(Trie, pattern+'$', number_of_nodes, i
    ↪ )

  return Trie
```

29.3 Add To Trie

Funkcija ima četiri parametra, `Trie`, `pattern`, `number_of_nodes` i `pattern_id`. Funkcija vraća izmenjeni `Trie` i broj čvorova.

Čvora sa oznakom 'root' smeštamo u promenljivu `current_node`. Sekvencu prolazimo karakter po karakter, svaki ćemo smestiti u promenljivu `c`. Ukoliko se taj karakter nalazi u mapi trenutnog čvora, onda trenutni čvor postaje vrednost koja odgovara karakteru `c` iz mape čvora `current_node`. U suprotnom, ukoliko nismo došli do kraja sekvence, odnosno, ukoliko je `c` različit od '$' onda pravimo novi čvor.

Pravimo čvor sa oznakom `i` na koju nadovezujemo trenutni broj čvorova. Njegovu mapu postavljamo na praznu. Vrednost koja odgovara karakteru `c` iz mape čvora `current_node` postavljamo na oznaku novog čvora. Trenutni čvor dobija vrednost novog čvora i uvećavamo broj čvorova za jedan.

Ukoliko smo došli do kraja sekvence, pravimo novi čvor sa oznakom `pattern_id` i postavljamo ga za vrednost koja odgovara karakteru `c` iz mape čvora `current_node`, a mapa novog čvora inicijalizovana je na praznu.

```
def add_to_trie(Trie, pattern, number_of_nodes, pattern_id):
  current_node = 'root'

  for c in pattern:
    if c in Trie[current_node]:
      current_node = Trie[current_node][c]
    else:
      if c != '$':
        Trie['i'+str(number_of_nodes)] = {}
        Trie[current_node][c] = 'i'+str(number_of_nodes)
        current_node = 'i'+str(number_of_nodes)
        number_of_nodes += 1
      else:
        Trie[current_node][c] = pattern_id
        Trie[pattern_id] = {}

  return (Trie, number_of_nodes)
```

29.3.1 Test primer

`patterns` = ['ananas', 'and', 'antenna', 'banana', 'bandana', 'nab', 'nana', 'pan']
`query` = 'bananananaspand'
`Trie = trie_construction(patterns)`
`found_patterns` =

30 Pattern Matching With Suffix Array

Funkcija ima dva parametra, `suffix_array` - sufiksni niz napravljen na osnovu teksta, i `pattern` - sekvenca koju tražimo u tekstu. Povratna vrednost je lista pozicija na kojima počinje sekvenca u tekstu.

Održavamo vrednosti `top`, koja kreće od pozicija 0, i `bottom`, koja kreće od pozicije |suffix_array|-1. Dok važi da je `top` manji od `bottom` izračunavamo srednji element `mid`, kao srednju vrednost ove dve.

Ukoliko je dužina sufiksa središnjeg veća od dužine sekvence koju tražimo, onda proveravamo i da li je prefiks dužine |pattern| u središnjem sufiksu jednak sekvenci `pattern`. Ako jeste, vraćamo povratnu vrednost funkcije `find_neighborhood` (30.1).

Sledeći korak odnosi se na ažuriranje vrednosti za `top` i `bottom` kao u binarnoj pretrazi. Tako da, ako je sekvenca manja od središnjeg elementa sufiksnog niza, onda se `bottom` pomera na jedno mesto ispred `mid`, u suprotnom pomeramo `top` na mestoiza `mid`.

```
1  def pattern_matching_with_suffix_array(suffix_array, pattern):
2    top = 0
3    bottom = len(suffix_array)-1
4
5    while top <= bottom:
6      mid = (top + bottom) // 2
7
8      if len(suffix_array[mid]) > len(pattern):
9        if suffix_array[mid][:len(pattern)] == pattern:
10         return find_neighborhood(suffix_array, mid, pattern)
11
12     if pattern < suffix_array[mid]:
13       bottom = mid - 1
14     else:
15       top = mid + 1
```

30.1 Find Neighborhood

Funkcija ima tri parametra, `suffix_array`, `mid` i `pattern`. Povratna vrednost je lista pozicija na kojima počinje sekvenca `pattern`.

Krećemo od središnjeg elementa i pomeramo se odozgo nadole i odozdo nagore dok su uspunjeni odgovarajući uslovi.

Za kretanje odozgo naniže bitno je da brojač ostane pozitivna vrednost, da dužina gornjeg elementa sufiksnog niza bude veća od dužine sekvence koju tražimo i da je prefiks dužine |pattern| gornje niske u sufiksnom nizu jednaka sekvenci `pattern`.

Za kretanje odozdo naviše uslovi su da brojač ne pređe dužinu niza, da je dužina donjeg elementa sufiksnog niza veća od dužine sekvence i da je prefiks dužine |pattern| donje niske u sufiksnom nizu jednaka sekvenci `pattern`.

Kada su obe granice određene, postavljamo ih u petlji. Krećemo od gornje granice pomerene za 1 i idemo do donje. U listu pozicija (koja je inicijalno prazna) dodajemo razliku dužine sufiksnog niza i dužine elementa na poziciji `i`.

Pre nego što vratimo listu, sortiramo je rastuće.

```
def find_neighborhood(suffix_array, mid, pattern):
  up = mid
  down = mid

  while up >= 0 and len(suffix_array[up]) > len(pattern) and suffix_array[up][:
    ↪ len(pattern)] == pattern:

    up -= 1

  while down < len(suffix_array) and len(suffix_array[down]) > len(pattern) and
    ↪  suffix_array[down][:len(pattern)] == pattern:

    down += 1

  positions = []

  for i in range(up+1, down):
    positions.append(len(suffix_array) - len(suffix_array[i]))

  positions.sort()
  return positions
```

30.2 Suffix Array Construction

Funkcija ima jedan parametar, `string`. Povratna vrdnost je sufiksni niz napravljen na osnovu niske `string`.

Sufiksni niz na početku je prazna lista, a niski nadovezujemo '$' na kraj. Nisku prolazimo pomoću indeksa `i`, a u svakoj iteraciji u sufiksni niz dodajemo podnisku koja počinje na poziciji `i`. Niz sortiramo i vraćamo iz funkcije.

```
def suffix_array_construction(string):
  suffix_array = []
  string += '$'

  for i in range(len(string)):
    suffix_array.append(string[i:])

  suffix_array.sort()
  return suffix_array
```

30.2.1 Test primer

`string` = 'ananas'
`suffix_array = suffix_array_construction(string)`
`pattern` = 'nana'
`found_patterns` =

U nastavku su opisane modifikacije ovih funkcija tako da rade kada umesto jedne niske imamo niz niski.

31 Pattern Matching With Suffix Array

Funkcija ima dva parametra, `suffix_array` - sufiksni niz napravljen na osnovu teksta, i `pattern` - sekvenca koju tražimo u tekstu. Povratna vrednost je lista pozicija na kojima počinje sekvenca u tekstu.

Obratiti pažnju da sufikni niz više nije niz niski već niz torki. Na prvom mestu nalazi se sufiks, na drugom je redni broj niske čiji je to sufiks, a na trećem je indeks na kom počinje sufiks u niski.

Održavamo vrednosti `top`, koja kreće od pozicija 0, i `bottom`, koja kreće od pozicije |suffix_array|-1. Dok važi da je `top` manji od `bottom` izračunavamo srednji element `mid`, kao srednju vrednost ove dve.

Ukoliko je dužina sufiksa središnjeg veća od dužine sekvence koju tražimo, onda proveravamo i da li je prefiks dužine |pattern| u središnjem sufiksu jednak sekvenci `pattern`. Ako jeste, vraćamo povratnu vrednost funkcije `find_neighborhood` (31.1).

Sledeći korak odnosi se na ažuriranje vrednosti za `top` i `bottom` kao u binarnoj pretrazi. Tako da, ako je sekvenca manja od središnjeg elementa sufiksnog niza, onda se `bottom` pomera na jedno mesto ispred `mid`, u suprotnom pomeramo `top` na mestoiza `mid`.

```
1  def pattern_matching_with_suffix_array(suffix_array, pattern):
2    top = 0
3    bottom = len(suffix_array)-1
4
5    while top <= bottom:
6      mid = (top + bottom) // 2
7
8      if len(suffix_array[mid][0]) > len(pattern):
9        if suffix_array[mid][0][:len(pattern)] == pattern:
10         return find_neighborhood(suffix_array, mid, pattern)
11
12     if pattern < suffix_array[mid][0]:
13       bottom = mid - 1
14     else:
15       top = mid + 1
```

31.1 Find Neighborhood

Funkcija ima tri parametra, `suffix_array`, `mid` i `pattern`. Povratna vrednost je lista pozicija na kojima počinje sekvenca `pattern`.

Krećemo od središnjeg elementa i pomeramo se odozgo nadole i odozdo nagore dok su uspunjeni odgovarajući uslovi.

Za kretanje odozgo naniže bitno je da brojač ostane pozitivna vrednost, da dužina niske gornjeg elementa sufiksnog niza bude veća od dužine sekvence koju tražimo i da je prefiks dužine |pattern| gornje niske u sufiksnom nizu jednaka sekvenci `pattern`.

Za kretanje odozdo naviše uslovi su da brojač ne pređe dužinu niza, da je dužina niske donjeg elementa sufiksnog niza veća od dužine sekvence i da je prefiks dužine |pattern| donje niske u sufiksnom nizu jednaka sekvenci `pattern`.

Kada su obe granice određene, postavljamo ih u petlji. Krećemo od gornje granice pomerene za 1 i idemo do donje. U listu pozicija (koja je inicijalno prazna) dodajemo par indeksa - redni broj niske (tj. drugi element torke elementa sufiksnog niza) i indeks na kom počinje sufiks (tj. treći element torke elementa sufiksnog niza) elementa na poziciji `i`.

Pre nego što vratimo listu, sortiramo je rastuće.

```
1  def find_neighborhood(suffix_array, mid, pattern):
```

```
  up = mid
  down = mid

  while up >= 0 and len(suffix_array[up][0]) > len(pattern) and suffix_array[up
    ↪ ][0][:len(pattern)] == pattern:

    up -= 1

  while down < len(suffix_array) and len(suffix_array[down][0]) > len(pattern)
    ↪ and suffix_array[down][0][:len(pattern)] == pattern:

    down += 1

  positions = []

  for i in range(up+1, down):
    positions.append((suffix_array[i][1], suffix_array[i][2]))

  positions.sort()
  return positions
```

31.2 Suffix Array Construction

Funkcija ima jedan parametar, `strings`. Povratna vrdnost je sufiksni niz napravljen na osnovu niza niski `strings`.

Sufiksni niz na početku je prazna lista, a gradićemo ga tako da elementi budu torke, a ne sami sufiksi. Jedna torka sadrži sufiks jedne niske iz niza, redni broj te niske i poziciju na kojoj počinje sufiks u niski.

Redom indeksiramo elemente niza (indeksom `s`), i svakom na kraj nadovezujemo '$'. Zatim, nisku prolazimo pomoću indeksa `i`, a u svakoj iteraciji u sufiksni niz dodajemo torku koja sadrži podnisku niske koja počinje na poziciji `i`, poziciju niske u nizu tj. `s` i indeks na kom počinje sufiks, odnosno, `i` . Niz sortiramo i vraćamo iz funkcije.

```
def suffix_array_construction(strings):
  suffix_array = []

  for s in range(len(strings)):
    string = strings[s] + '$'
    for i in range(len(string)):
      suffix_array.append((string[i:],s, i))

  suffix_array.sort()
  return suffix_array
```

31.2.1 Test primer

`strings` = ['ananas', 'and', 'antenna', 'banana', 'bandana', 'nab', 'nana', 'pan']
`suffix_array = suffix_array_construction(strings)`
`pattern` = 'an'
`found_patterns` =

32 BW Matching

Funkcija ima tri parametra, `fisrt_column`, `last_column` i `pattern`. Povratna vrednost je pozicija na kojoj počinje sekvenca `pattern` u tekstu.

Održavamo vrednosti `top`, koja kreće od pozicija 0, i `bottom`, koja kreće od pozicije |last_column|-1. Petlja se vrti dok važi da je `top` manji od `bottom`. Ako sekvenca ima još karaktera, izdvajamo poslenji simbol u `symbol` a `pattern` skraćujemo za posledni karakter. Izdvajamo podskup poslednje kolone na intervalu `[top, bottom]` u `subset`.

Metodom `index` proveravamo da li `subset` sadrži karakter `symbol` (funkcija vraća indeks, ako pronađe element, odnosno 0, ako ga ne nađe).

Ukoliko ga sadrži, želimo da odredimo `top_index` i `bottom_index`, koje incijalizujemo na -1. Petljom prolazimo poslednju kolonu, od `top` do `bottom` + 1. Ukoliko je `symbol` jednak karakteru poslednje kolone, menjamo `top_index` i `bottom_index`. Gornji indeks menjamo samo ako već nije dobio neku vrednost, odnosno, ako i dalje ima vrednost -1. U tom slučaju ga postavljamo na vrednost indeksa petlje, a donji indeks u svakom slučaju dobija tu vrednost.

Sada želimo da izbrojimo koliko se karaktera u poslednjoj koloni poklapa sa `symbol`, do gornjeg indeksa (`top_count`) i do donjeg indeksa (`bottom_count`). To vršimo jednostavnim prolaskom poslednje kolone i upoređivanjem karaktera.

Treba još ažurirati `top` i `bottom`. Oba ažuriramo pomoću funkcije `last_to_first` (32.1), s tim da prosleđujemo različite elemente poslednje kolone i brojače.

Ukoliko `subset` ipak ne sadrži `symbol`, vraćamo -1.

Ukoliko Nema više karaktera u sekvenci `pattern`, vraćamo razliku `bottom` i `top` uvećanu za jedan.

```
1  def bw_matching(first_column, last_column, pattern):
2    top = 0
3    bottom = len(last_column) - 1
4
5    while top <= bottom:
6      if len(pattern) > 0:
7        symbol = pattern[-1]
8        pattern = pattern[:-1]
9
10       subset = last_column[top:bottom+1]
11
12       if subset.index(symbol) != -1:
13         top_index = -1
14         bottom_index = -1
15
16         for i in range(top, bottom+1):
17           if symbol == last_column[i]:
18             if top_index == -1:
19               top_index = i
20             bottom_index = i
21
22         top_count = 0
23
24         for i in range(top_index+1):
25           if last_column[i] == symbol:
26             top_count += 1
27
28         bottom_count = top_count
29
```

```
30          for i in range(top_index+1, bottom_index + 1):
31            if last_column[i] == symbol:
32              bottom_count += 1
33
34          top = last_to_first(first_column, last_column[top_index], top_count)
35          bottom = last_to_first(first_column, last_column[bottom_index],
    ↪ bottom_count)
36
37        else:
38          return 0
39
40      else:
41        return bottom - top + 1
```

32.1 Last To First

Funkcija ima tri parametra, `first_column`, karakter `c` i broj pojavljivanja tog karaktera u poslednjoj koloni `count`. Povratna vrednost je poslednji indeks na kom počinje taj karakter u prvoj koloni.

Dakle, prolazimo redom sve karaktere prve kolone. Ukoliko je `c` jednak karakteru prve kolone na poziciji `i` onda u zavisnosti od broja pojavljivanja radimo sledeće. Ako je broj pojavljivanja jedan 1, vraćamo indeks `i`. Inače, umanjujemo brojač za jedan.

```
1  def last_to_first(first_column, c, count):
2    for i in range(len(first_column)):
3      if first_column[i] == c:
4        if count == 1:
5          return i
6        count -= 1
```

32.2 BWT

Funkcija ima jedan parametar, nisku `s`. Povratna vrednost poslednja kolona matrice permutacija niske `s`.

Polazimo od prazne matrice, koja je predstavljena kao niz niski. Na kraj niske `s` nadovezujemo karakter '$'. Petlja se izvršava `|s|` puta. U svakoj iteraciji, u matricu andovezujemo nisku `s` a zatim je ciklično pomeramo za jedno mesto u levo. Nakon petlje, sortiramo matricu i vraćamo listu poslednjih karaktera iz svakog reda matrice, odnosno, poslednju kolonu.

```
1  def BWT(s):
2    matrix = []
3    s += '$'
4
5    for i in range(len(s)):
6      matrix.append(s)
7      s = s[1:] + s[0]
8
9    matrix.sort()
10   return [row[-1] for row in matrix]
```

32.2.1 Test primer

`s` = 'panamabananas'
`last_column = BWT(s)`
`pattern` = 'ana'
`first_column = last_column[:]`
`found_patterns` =

33 HMM

Funkcija ima dva argumenta, listu niski `strings_pos` i `string_neg` (pozitivno i negativno, kao kod primera sa bacanjem pristrasnog, što je označavalo negativne niske, i fer novčića, što je označavalo pozitivne niske). Povratna vrednost je HMM dijagram.

HMM na početku je prazna mapa. Ključevi u mapi su niske sastavljene od karaktera i znaka '+' ili '-', a vrednosti su mape. Svaka od tih mapa sadrži tri ključa: `'state'` - 1 -pozitivni, 0 - negativni; `'nucleotide'` i `'transitions'` - mapa prelazaka (ključevi su karakteri, a vrednosti su brojevi).

Prvo prolazimo listu pozitivnih stringova, element po element, a onda i negativne. [1] Svaku nisku prolazimo pomoću indeksa `i`. Pamtimo dve promenljive, `c_prev_state`, koja dobija vrednost karaktera na prethodnoj poziciji na koji nadovezujemo znak '+' (odnosno, '-' za negativne niske), i `c_curr` koja označava trenutni karakter u niski.

Ukoliko se prethodni karakter ne nalazi u `HMM` dodajemo ga sa praznom mapom. Zatim mu inicijalizujemo `state` na 1 (odnosno, 0 za negativne). Vrednost za ključ `nucleotide` je prethodni karakter niske, a `transition` dobija praznu mapu.

Ukoliko se trenutni karakter ne nalazi u tranzicijama `c_prev_state`, dodajemo ga i inicijalizujemo mu vrednost na 0. U svakom slučaju ćemo uvećati tu vrednost za 1.

Sada prolazimo sve ključeve u `HMM`. Računamo sumu vrednosti svih tranzicija. Zatim, vrednost svake tranzicije delimo tom sumom i množimo sa 0.98. Pre prelaska na sledeću iteraciju, dodajemo još jedan element u mapu, a to je ('0', 0.02).

```
1  def HMM(strings_pos, strings_neg):
2
3    HMM = {}
4
5    for string in strings_pos:
6      for i in range(1, len(string)):
7        c_prev_state = string[i-1] + '+'
8        c_curr = string[i]
9
10       if c_prev_state not in HMM:
11         HMM[c_prev_state] = {}
12         HMM[c_prev_state]['state'] = 1
13         HMM[c_prev_state]['nucleotide'] = string[i-1]
14         HMM[c_prev_state]['transitions'] = {}
15
16       if c_curr not in HMM[c_prev_state]['transitions']:
17         HMM[c_prev_state]['transitions'][c_curr] = 0
18
19       HMM[c_prev_state]['transitions'][c_curr] += 1
20
21   for string in strings_neg:
22     for i in range(1, len(string)):
23       c_prev_state = string[i-1] + '-'
24       c_curr = string[i]
25
26       if c_prev_state not in HMM:
27         HMM[c_prev_state] = {}
28         HMM[c_prev_state]['state'] = 0
29         HMM[c_prev_state]['nucleotide'] = string[i-1]
```

[1] Pošto je posao za pozitivne i negativne većinom isti, opisaćemo obe petlje zajedno, a delove koji se razlikuju opisujemo u zagradama za negativne niske.

```
30          HMM[c_prev_state]['transitions'] = {}
31
32        if c_curr not in HMM[c_prev_state]['transitions']:
33          HMM[c_prev_state]['transitions'][c_curr] = 0
34
35        HMM[c_prev_state]['transitions'][c_curr] += 1
36
37    for source in HMM:
38      output_sum = 0
39      for dest in HMM[source]['transitions']:
40        output_sum += HMM[source]['transitions'][dest]
41
42      for dest in HMM[source]['transitions']:
43        HMM[source]['transitions'][dest] = (HMM[source]['transitions'][dest] /
      ↪ output_sum) * 0.98
44
45      HMM[source]['0'] = 0.02
46
47
48    return HMM
```

34 Viterbi

Funkcija ima dva parametra, `HMM` i `string`. Povratna vrednost je putanja `path`.

Putanja je na početku prazna niska, a pored toga održavamo listu, na početku praznih, mapa, po jednu za svaki karakter niske.

Pomoću indeksa `i` redom prolazimo karaktere niske. Izdvajamo trenutni karakter u promenljivu `nucleotide`. U svakoj iteraciji, osim prve, u putanju dodajemo najveće stanje prelaska. Promenljive `max_transition_prob` i `max_transition_state` postavljamo na -1, odnosno, na praznu nisku.

Za svako stanje u `HMM` proveravamo da li je njegov nukleotid jednak trenutnom nukleotidu. Ukoliko nije, u `matrix` postavljamo 0 za `i`-ti element i trenutno stanje. Ukoliko jeste, u prvoj iteraciji postavljamo tu vrednost na 0.125, a u svim ostalim iteracijama radimo sledeće.

Prvo, postavljamo promenljive `max_prev` na -1 i `max_prev_state` na praznu nisku. Za svako prethodno stanje u `matrix` prethodnog elementa proveravamo da li se prošlo i trenutno stanje poklapaju. Ako da, postavljamo `state_change_prob` na 0.99, inače na 0.01. Zatim, ako se nukleotid trenutnog stanja nalazi u tranzicijama prethodnog stanja onda `transition_prob` dobija vrednost tranzicije prethodnog stanja za nukleotid trenutnog stanja, a inače 0.

Još uvek obrađujemo treći slučaj, a sledeći korak jeste da odredimo vrednost trenutnog stanja. Promenljiva `curr_state` dobija vrednost proizvoda vrednosti prethodnog stanja `i-1`-og elementa liste `matrix` sa `state_change_prob` i `transition_prob`. Ukoliko je trenutno stanje veće od `max_prev`, ažuriramo maksimume, tako da `max_prev` dobije vrednost `curr_state`, a `max_prev_state` dobije vrednost `prev_state`.

Vrednost za ključ `state` `i`-tog elementa liste dobija vrednost `max_prev` i ostaje još da uporedimo `max_prev` i `max_transition_prob`. Ukoliko je veće ažuriramo maksimume tako da `max_transition_prob` dobije vrednost `max_prev`, a `max_transition_state` dobije vrednost `max_prev_state`.

Sada su obrađeni svi karakteri niske i treba da odredimo završno stanje. Promenljiva `max_end` dobija vrednost -1, a `max_end_state` je prazna niska. Prolazimo sva stanja poslednjeg elementa liste i tražimo ono sa najvećom vrednošću.

Na putanju nadovezujemo `max_end_state` i vraćamo putanju.

```
def viterbi(HMM, string):

  matrix = [{} for i in range(len(string))]
  path = ""

  for i in range(len(string)):
    nucleotide = string[i]

    if i > 0:
      path += max_transition_state

    max_transition_prob = -1
    max_transition_state = ''

    for state in HMM:

      if HMM[state]['nucleotide'] != nucleotide:
        matrix[i][state] = 0

      elif i == 0:
        if HMM[state]['nucleotide'] == nucleotide:
          matrix[i][state] = 0.125

      else:
        max_prev = -1
        max_prev_state = ''

        for prev_state in matrix[i-1]:
          if HMM[prev_state]['state'] == HMM[state]['state']:
            state_change_prob = 0.99
          else:
            state_change_prob = 0.01

          if HMM[state]['nucleotide'] in HMM[prev_state]['transitions']:
            transition_prob = HMM[prev_state]['transitions'][HMM[state]['
    ↪ nucleotide']]
          else:
            transition_prob = 0

          curr_state = matrix[i-1][prev_state] * transition_prob *
    ↪ state_change_prob

          if curr_state > max_prev:
            max_prev = curr_state
            max_prev_state = prev_state

        matrix[i][state] = max_prev

        if max_prev > max_transition_prob:
          max_transition_prob = max_prev
          max_transition_state = max_prev_state

```

```
51    max_end = -1
52    max_end_state = ''
53
54    n = len(string)
55    for state in matrix[n-1]:
56      if matrix[n-1][state] > max_end:
57        max_end = matrix[n-1][state]
58        max_end_state = state
59
60    path += max_end_state
61
62    return path
```

34.1 Test primer

```
strings_pos = [
   'ACACAGACGCACA',
   'CACATAGACAGGCATACACA',
   'AAATACAGTATCTTTGCACTCCCGGAGTGCGG'
]
strings_neg = [
   'CGAGCGTGTGAGTGAGAGATGAG',
   'GTGGAACAGTAGGTAGGAGAGTG',
   'AAATACAGTATCTTTGCACTCCCGGAGTGCGG']
model = HMM(strings_pos, strings_neg)
path =
```