# Bioinformatika

18. april 2018.

# Sadržaj

# Predgovor

Tekst se sastoji od proširenih beleški sa predavanja na osnovu knjige Pavel A. Pevzner, Phillip Compeau: Bioinformatics Algorithms: An Active Learning Approach.
Tekst su sastavili studenti sa kursa održanog u školskoj 2017/2018 godini:

- Una Stanković 1095/2016
- Marina Nikolić 1055/2017
- Strahinja Milojević 1049/2017
- Anja Bukurov 1082/2016
- Nikola Ajzenhamer 1083/2016
- Vojislav Stanković 1080/2016
- Milica Đurić 1084/2016
- Ana Stanković 1096/2016

# Glava 1

# Gde u genomu počinje replikacija genoma?

## 1.1 Uvod

Na samom početku, želimo da definišemo pojam bioinformatike i da pokušamo da shvatimo koji je njen osnovni cilj. Da bismo to postigli, pogledajmo tri definicije, iz različitih izvora:

- "Bioinformatika je nauka koja se bavi prikupljanjem i analizom kompleksnih bioloških podataka poput genetskih kodova." - Oksfordski rečnik (engl. *Oxford Dictionary*)
- "Bioinformatika predstavlja prikupljanje, klasifikaciju, čuvanje i analizu biohemijskih i bioloških informacija korišćenjem računara, a posebno se primenjuje u molekularnoj genetici i genomici." - Rečnik Meriam-Vebster (engl. *Merriam-Webster Dictionary*)
- "Bioinformatika je interdisciplinarno polje koje radi na razvoju metoda i softverskih alata za razumevanje bioloških podataka." - Vikipedija (engl. *Wikipedia*)

Na osnovu ove tri definicije možemo zaključiti da:
*Bioinformatika predstavlja primenu računarskih tehnologija u istraživanjima u oblasti biologije i srodnih nauka.*

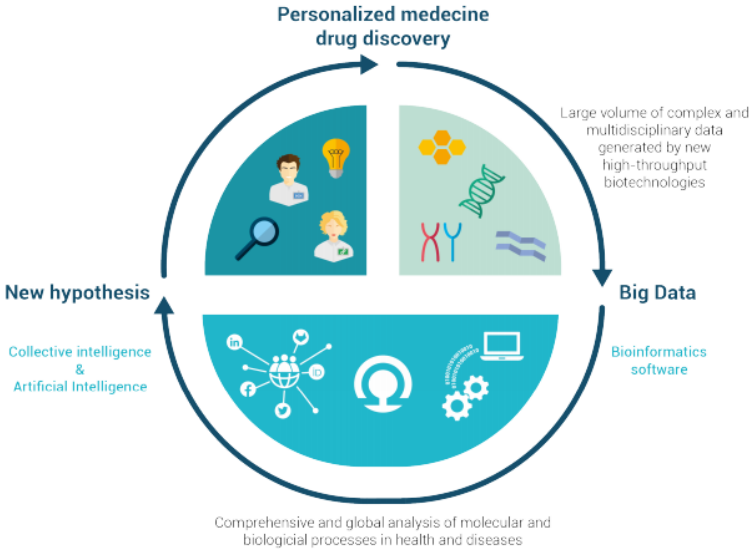
Bioinformatika ima široku primenu i njene primene rastu zajedno sa razvojem discipline. Kao što možemo videti na slici ispod, primena bioinformatike se može sagledati kroz personalizovanu medicinu. Naime, na osnovu prikupljene veće količine podataka i njihove analize, uz pomoć različitih računarskih metoda, na primer metoda veštačke inteligencije, možemo doći do informacija potrebnih da na najbolji način lečimo pacijenta ili mu odredimo terapiju koja će mu na najbolji, najbrži i najbezbolniji način pomoći da prevaziđe određene zdravstvene probleme.

Bioinformatika je spoj više različitih disciplina, kao što su:

- Statistika
- Istraživanje podataka
- Računarstvo
- Računarska biologija
- Biologija
- Biostatistika

Prikaz preklapanja ovih disciplina možemo videti na slici 1.2.

Slika 1.1: Primena bioinformatike




## 1.2 Replikacija genoma

### 1.2.1 DNK

Dezoksiribonukleinska kiselina (akronimi DNK ili DNA, od engl. *deoxyribonucleic acid*), nukleinska kiselina koja sadrži uputstva za razvoj i pravilno funkcionisanje svih živih organizama. Zajedno sa RNK i proteinima, DNK je jedan od tri glavna tipa makromolekula koji su esencijalni za sve poznate forme života.

Sva živa bića svoj genetički materijal nose u obliku DNK, sa izuzetkom nekih virusa koji imaju ribonukleinsku kiselinu (RNK). DNK ima veoma važnu ulogu ne samo u prenosu genetičkih informacija sa jedne na drugu generaciju, već sadrži i uputstva za građenje neophodnih ćelijskih organela, proteina i RNK molekula. DNK segment koji sadrži ova važna uputstva se naziva gen.

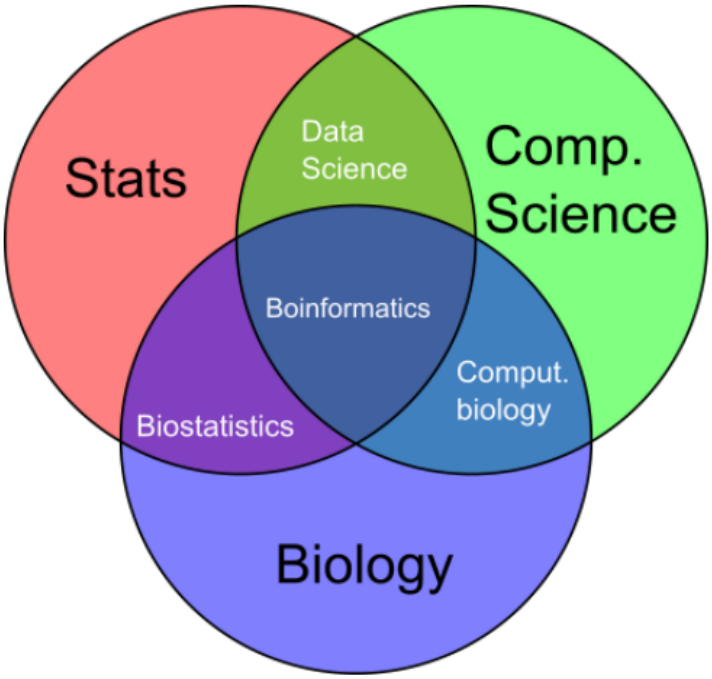
DNK se sastoji iz dva polimerna lanca koji imaju antiparalelnu orijentaciju, i svaki od njih je sastavljen od azotnih baza:

- adenin (A)
- timin (T)
- guanin (G)
- citozin (C)

Lanci DNK su međusobno spojeni i to tako da se veze uspostavljaju isključivo između adenina i citozina ili između guanina i timina. Na osnovu toga, ako nam je poznat sastav jednog lanca, lako možemo zaključiti i sastav drugog lanca, zbog čega se kaže da su DNK lanci **međusobno komplementarni**.

Da bismo lakše manipulisali sa informacijama koje DNK nosi i približili sadržaj računarskoj struci, DNK ćemo posmatrati kao nisku nad azbukom $A, C, G, T$.

Slika 1.2: Preklapanjem različitih disciplina dobijamo bioinformatiku.




### 1.2.2 Replikacija genoma u ćeliji

Replikacija genoma je jedan od najvažnijih zadataka ćelije. Pre nego što se podeli, ćelija mora da najpre replicira svoj genom, tako da svaka od ćerki ćelija dobije svoju kopiju.

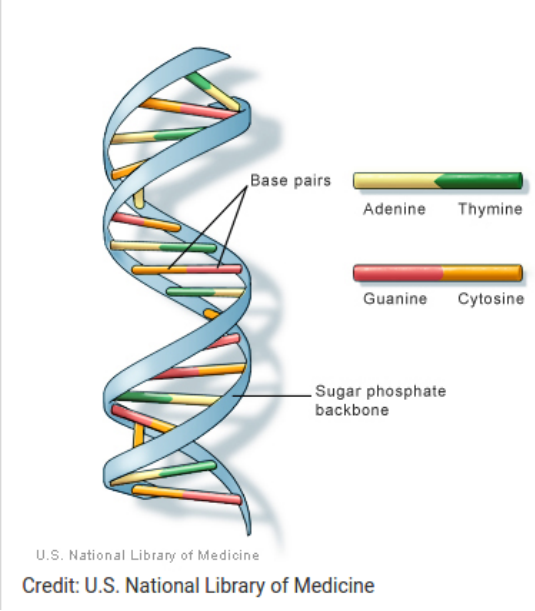
Dzejms Votson (engl. *James Watson*) i Fransis Krik (engl. *Fransis Crick*) su 1953. godine napisali rad u kome su primetili da postoji mehanizam za kopiranje genetskog materijala. Oni su uočili da se lanci roditeljskog DNK molekula odvijaju tokom replikacije i da se, potom, svaki lanac ponaša kao uzorak za sintezu novog lanca (na osnovu toga što se uvek spajaju iste aminokiseline A-C i G-T, rekreiranje lanca je moguće). Kao rezultat ovakvog ponašanja, proces replikacije počinje parom komplementarnih lanca i završava se sa dva para komplementarnih lanaca, kao što se može videti na slici ispod.

Replikacija počinje u regionu genoma koji se naziva **početni region replikacije** (skraćeno *oriC*), izvode je enzimi koje se nazivaju DNK polimeraze, koje predstavljaju mašine za kopiranje na molekularnom nivou.

Nalaženje početnog regiona replikacije predstavlja veoma važan problem, ne samo za razumevanje funkcionisanja kako se ćelije repliciraju, već je koristan i u raznim biomedicinskim problemima. Na primer, neki metodi genskih terapija uključuju genetski izmenjene mini genome, koji se zovu virusni vektori, zbog svoje sposobnosti da prodru kroz ćelijski zid (poput pravih virusa). Virusni vektori u sebi nose veštačke gene koji unapredjuju postojeći genom. Genska terapija je prvi put uspešno izvršena 1990. godine na devojčici koja je bila toliko otporna na infekcije da je bila primorana da živi isključivo u sterilnom okruženju.

Osnovna ideja genske terapije je da se pacijent, koji pati od nedostatka nekog bitnog gena, zarazi viralnim vektorom koji sadrži veštački gen koji enkodira terapeutski protein. Jednom kad

Slika 1.3: Prikaz DNK, slika preuzeta sa https://ghr.nlm.nih.gov/primer/basics/dna




je unutar ćelije, vektor se replicira, što dovodi do lečenja bolesti pacijenta. Da bi moglo da dodje do ovoga, biolozima je neophodno da znaju gde je *oriC*.

### Kako ćelija prepoznaje *oriC*?

Pitamo se kako ćelija prepoznaje oriC? Sigurno je da postoji neka niska aminokiselina koja označava oriC, ali kako ga prepoznati?

Ograničimo se na bakterijski genom, koji se sastoji od jednog kružnog hromozoma. Istraživanje je pokazalo da je region, koji predstavlja oriC kod bakterija, dug svega nekoliko stotina nukleotida.

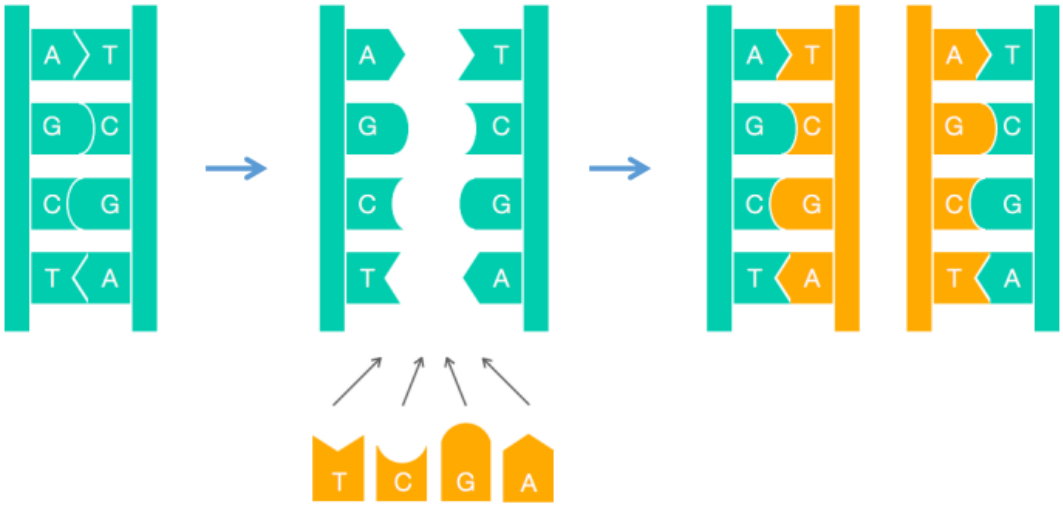
Poznato je da DNKA utiče na početak replikacije. *DNKA* je protein koji se vezuje na kratki segment unutar oriC, poznatiji kao **DNKA boks**. Ona predstavlja poruku unutar sekvence DNK koja govori proteinu DNKA da se veže baš tu. Postavlja se pitanje kao pronaći taj region bez prethodnog poznavanja izgleda DNKA boks?

Da bismo bolje razumeli *problem skrivene poruke* uzmimo za primer priču Edgara Alana Poa - "Zlatni jelenak" (engl. "*The Gold-Bug*"). Naime, u toj priči jedan od likova, Vilijam Legrand (engl. *William Legrand*), treba da dešifruje poruku :

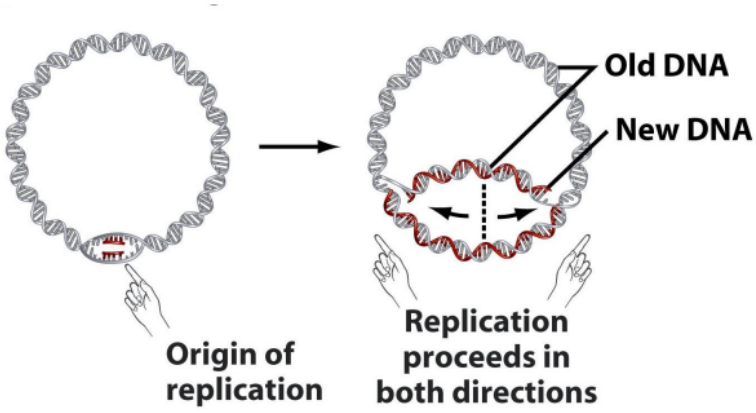
```
53++!305))6*;4826)4+.)4+);806*;48!8‘6 0))85;]8*:+*8!83(88)5*!;46(;88*96*?;8
)*+(;485);5*!2:*+(;4956*2(5*4)8‘8*;40 69285);)6!8)4++;1(+9;48081;8:8+1;48!8 5;4)
485!528806*81(+9;48;(88;4(+?34;48 )4+;161;:188;+?;
```

On uočava da se ";48" pojavljuje veoma često, i da verovatno predstavlja "THE", najčešću reč u engleskom jeziku. Znajući to, zamenjuje karaktere odgovarajućim slovima i postepeno dešifruje celu poruku.

Slika 1.4: Prikaz replikacije




Slika 1.5: Prikaz početka replikacije kod bakterija




```
53++!305))6*THE26)H+.)H+)806*THE !E‘60))E5;]E*:+*E!E3(EE)5*!TH6(T EE*96*?;E)*+
(THE5)T5*!2:*+(TH956 *2(5*H)E‘E*TH0692E5)T)6!E)H++T1( +9THE0E1TE:E+1
THE!E5T4)HE5!52880 6*E1(+9THET(EETH(+?34THE)H+T161T :1EET+?T
```

Želeli bismo da ovaj princip primenimo na naš problem nalaska *oriC*-a. Ideja je da uvidimo da li postoje reči koje se neuobičajeno često pojavljuju. Uvedimo termin k-gram da označimo string dužine $k$ i COUNT(Text, Pattern) da označimo broj puta kojih se k-gram *Pattern* pojavio u tekstu *Text*. Osnovna ideja je da pomeramo prozor, iste dužine kao k-gram *Pattern*, niz tekst, usput proveravajući da li se pojavljuje *Pattern* u nekome od njih.

```
1  PATTERNCOUNT(Text, Pattern)
2      count = 0
3      for i = 0 to |Text| - |Pattern|
4          if Text(i,|Pattern|) = Pattern
5              count = count + 1
6      return count
```

Za neki *Pattern* kažemo da je on *najčešći k-gram* u tekstu $Text$, ako je njegov $COUNT$ najveći među svim k-gramima. Na primer, **ACTAT** je najčešći 5-gram u tekstu $Text$ = ACA**ACTAT**GCA**ACTAT**CGGGACA**ACTAT**CCT, a **ATA** je najčešći 3-gram u $Text$ = CG**ATATA**TCC**ATA**G.

Sada, problem pronalaska čestih reči možemo posmatrati kao računarski problem:

**Problem čestih reči:** Pronaći najčešće k-grame u niski karaktera.
*Ulaz:* Niska Text i ceo broj k.
*Izlaz:* Svi najčešći k-grami u niski Text.

Osnovni algoritam za pronalazak čestih k-grama u stringu $Text$ proverava sve k-grame koji se pojavljuju u tom stringu (takvih k-grama ima $|Text| - k + 1$) i potom izračunava koliko puta se svaki k-gram pojavljuje. Da bismo implementirali ovaj algoritam, moramo da izgenerišemo niz $COUNT$, gde je $COUNT(i) = COUNT(Text, Pattern)$ za $Pattern = Text(i, k)$.

```
1  FrequentWords(Text, k)
2      FrequentPatterns <- an empty set
3      for i = 0 to |Text| - k
4          Pattern <- the k-mer Text(i,k)
5          COUNT(i) <- PatternCount(Text, Pattern)
6      maxCount <- max value in array COUNT
7      for i = 0 to |Text| - k
8          if COUNT(i) = maxCount
9              add Text(i,k) to FrequentPatterns
10     remove duplicates from FrequentPatterns
11     return FrequentPatterns
```

Pitamo se, sada, kolika je složenost ovakvog pristupa?
Ovaj algoritam, iako uspešno nalazi ono što se od njega traži, nije najefikasniji. S obzirom na to da svaki k-gram zahteva $|Text| - k + 1$ provera, svaki od njih zahteva i do $k$ poređenja, pa je broj koraka izvršavanja funkcije $PatternCount(Text, Pattern)$ zapravo $(|Text| - k + 1) * k$. Osim toga, $FrequentWords$ mora pozvati $PatternCount$ $|Text| - k + 1$ puta (po jednom za svaki k-gram teksta), tako da je ukupan broj koraka $(|Text| - k + 1) * (|Text| - k + 1) * k$.
Iz navedenog, možemo zaključiti da je ukupna cena izvršavanja algoritma $FrequentWords$ $O(|Text|^2 * k)$.

### Primer: Pronalazak čestih reči kod bakterije *Vibrio cholerae*

Posmatrajmo, najpre, tablicu najčešćih k-grama u *oriC* regionu bakterije *Vibrio cholerae*. Da li nam se čini da se neki k-grami pojavljuju neuobičajeno često?

Na primer, 9-gram **ATGATCAAG** se pojavljuje tri puta u *oriC* regionu, da li nas to iznenađuje?

Označili smo najčešće 9-grame, umesto nekih drugih k-grama, jer je eksperimentima pokazano da su DNKA boksovi kod bakterija dugi 9 nukleotida. Uočimo da postoje četiri različita 9-grama koji se ponavljaju tri ili više puta u ovom regionu, to su: ATGATCAAG, CTTGATCAT, TCTTGATCA i CTCTTGATC.

Slika 1.6: Tablica najčešćih k-grama u *oriC* regionu bakterije Vibrio cholerae

| *k* | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|
| count | 25 | 12 | 8 | 8 | 5 | 4 | 3 |
| *k*-mers | tga | atga | gatca | tgatca | atgatca | atgatcaa | atgatcaag |
| | | | tgatc | | | | cttgatcat |
| | | | | | | | tcttgatca |
| | | | | | | | ctcttgatc |

Slika 1.7: Prikaz 9-grama ATGATCAAG i njegovog komplementa u *oriC* regionu Vibrio cholerae

atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaacctgagtgg
atgacatcaagataggtcgttgtatctccttcctctcgtactctcatgaccacggaaag**ATGATCAAG**
agaggatgatttcttggccatatcgcaatgaatacttgtgacttgtgcttccaattgacatcttcagc
gccatattgcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgttctgttt
atcttgttttgactgagacttgttaggatagacggtttttcatcactgactagccaaagccttactct
gcctgacatcgaccgtaaattgataatgaatttacatgcttccgcgacgatttacct**CTTGATCAT**cg
atccgattgaagatcttcaattgttaattctcttgcctcgactcatagccatgatgagct**CTTGATCA**
**T**gtttccttaaccctctatttttacggaaga**ATGATCAAG**ctgctgct**CTTGATCAT**cgtttc

Mala verovatnoća da se neki 9-gram toliko puta pojavi u *oriC*-u kolere, govori nam da neki od četiri 9-grama koje smo pronašli može biti potencijalni DNKA boks, koji započinje replikaciju. Ali, koji?

Podsetimo se da nukleotidi A i T, kao i C i G, su komplementarni. Ako imamo jednu stranu lanca DNK i neke slobodne nukleotide, možemo lako zamisliti sintezu komplementarnog lanca, kao što se vidi na slici ispod.

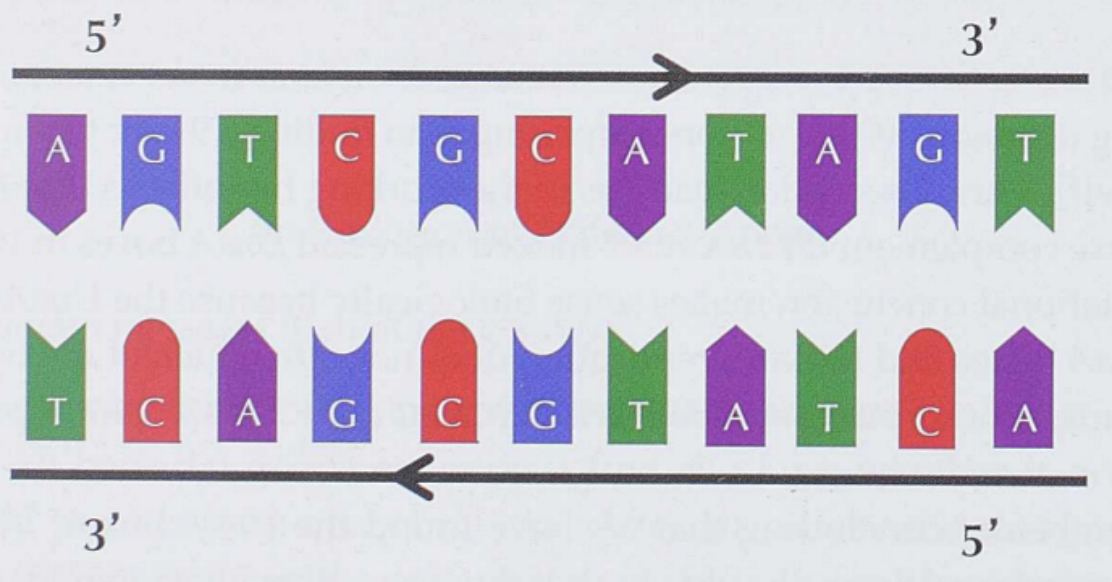
Posmatrajmo ponovo sliku 1.7. Na njoj možemo uočiti 6 pojavljivanja niski ATGATCAAG i CTTGATCAT, koji su zapravo komplementarni. Naći 9-gram koji se pojavljuje 6 puta u DNK nisci dužine 500 nukleotida, je još više iznenadjujuće, nego pronaći 9-gram koji se pojavljuje tri puta. Ovo posmatranje nas dovodi do toga da je ATGATCAAG (zajedno sa svojim komplementom) zaista DNKA boks Vibrio cholerae. Ovaj zaključak ima i smisla biološki, jer DNKA proteinu, koji se vezuje i započinje replikaciju, nije bitno za koji od dva lanca se vezuje.

### Primer: Pronalazak čestih reči kod bakterije *Thermotoga petrophila*

Nakon što smo pronašli skrivenu poruku za Vibrio cholerae, ne bi trebalo da odmah zaključimo da je ta poruka ista kod svih bakterija. Najpre bi trebalo da proverimo da li se ona nalazi u *oriC* regionu drugih bakterija, možda različite bakterije, imaju drugačije DNKA boksove. Uzmimo, za primer, *oriC* region bakterije Thermotoga petrophila. Ona predstavlja bakteriju koja obitava u izrazito toplim regionima, na primer u vodi ispod rezervi nafte, gde temperature prelaze 80 stepeni Celzijusa. Pogledajmo kako izgleda *oriC* region ove bakterije.

Možemo lako uočiti da se u ovom regionu nigde ne javljaju niske ATGATCAAG ili CTTGATCAT, iz čega zaključujemo da različite bakterije mogu koristiti različite DNKA boksove, kako bi pružile skrivenu poruku DNKA proteinu. Odnosno, za različite genome imamo različite DNKA boksove.

Slika 1.8: Komplementarni lanci se "kreću" u suprotnim smerovima.




Slika 1.9: Prikaz *oriC* regiona Thermotoga petrophila

```
aactctatacctccttttttgtcgaatttgtgtgatttatagagaaaatcttattaactgaaactaa
aatggtaggtttggtggtaggttttgtgtacattttgtagtatctgatttttaattacataccgta
tattgtattaaattgacgaacaattgcatggaattgaatatatgcaaaacaaacctaccaccaaac
tctgtattgaccatttaggacaacttcagggtggtaggtttctgaagctctcatcaatagactat
tttagtctttacaaacaatattaccgttcagattcaagattctacaacgctgttttaatgggcgtt
gcagaaaacttaccacctaaaatccagtatccaagccgatttcagagaaacctaccacttacctac
cacttacctaccacccgggtggtaagttgcagacattattaaaaacctcatcagaagcttgttcaa
aaatttcaatactcgaaacctaccacctgcgtcccctattatttactactactaataatagcagta
taattgatctgaaaagaggtggtaaaaaa
```

Najčešće reči u ovom *oriC* su:

- AACCTACCA,
- ACCTACCAC,
- GGTAGGTTT,
- TGGTAGGTT,
- AAACCTACC,
- CCTACCACC

Pomoću alata koji se zove Ori-Finder, nalazimo CCTACCACC i njegov komplement GGTGGTAGG kao potencijalne DNKA boksove naše bakterije. Ove dve niske se pojavljuju ukupno 5 puta.

Naučili smo da pronađemo skrivene poruke ako je *oriC* dat, ali ne znamo da pronađemo *oriC* u genomu.

Slika 1.10: Prikaz CCTACCACC i njenog komplementa u *oriC* regionu Thermotoga petrophila

aactctatacctccttttttgtcgaatttgtgtgatttatagagaaaatcttattaactgaaactaa
aatggtaggtttGGTGGTAGGttttgtgtacattttgtagtatctgatttttaattacataccgta
tattgtattaaattgacgaacaattgcatggaattgaatatatgcaaaacaaaCCTACCACCaaac
tctgtattgaccattttaggacaacttcagGGTGGTAGGtttctgaagctctcatcaatagactat
tttagtctttacaaacaatattaccgttcagattcaagattctacaacgctgttttaatgggcgtt
gcagaaaacttaccacctaaaatccagtatccaagccgatttcagagaaacctaccacttacctac
cacttaCCTACCACCcgggtggtaagttgcagacattattaaaaacctcatcagaagcttgttcaa
aaatttcaatactcgaaaCCTACCACCtgcgtcccctattatttactactactaataatagcagta
taattgatctgaaaagaggtggtaaaaaa

### 1.2.3 Pronalaženje početnog regiona replikacije

Zamislimo da pokušavamo da nađemo *oriC* u novom sekvenciranom genomu bakterije. Ako bismo tražili niske poput ATGATCAAG/CTTGATCAT ili CCTACCACC/GGTGGTAGG to nam verovatno ne bi bilo puno od pomoći, jer novi genom može koristiti potpuno drugačiju skrivenu poruku. Posmatrajmo, zato, drugačiji problem: umesto da tražimo grupe određenog k-grama, pokušajmo da nađemo svaki k-gram koji formira grupu u genomu. Nadajmo se da će nam lokacije ovih grupa u genomu pomoći da odredimo lokaciju *oriC*-a.
Ideja je da pomeramo prozor fiksirane dužine L kroz genom, tražeći region u kome se k-gram pojavljuje više puta uzastopno. Za L ćemo uzeti vrednost 500, koja predstavlja najčešću dužinu *oriC*-a kod bakterija.

Definisali smo k-gram kao *grupu*, ako se pojavljuje više puta unutar kratkog intervala u genomu. Formalno, k-gram *Pattern* formira $(L, t)$ grupu unutar niske *Genome* ako postoji interval genoma, dužine $L$, u kome se k-gram pojavljuje barem $t$ puta.

**Problem pronalaženja grupa.** Naći k-grame koji formiraju grupe unutar niske karaktera.
*Ulaz:* Niska Genome i celi brojevi k (dužina podniske), L (dužina prozora) i t (broj podniski u grupi).
*Izlaz:* Svi k-grami koji formiraju (L, t)-grupe u niski Genome.

U genomu bakterije E.coli postoji 1904 različitih 9- grama koji formiraju $(500, 3)$-grupe. Koji od njih ukazuje na početni region replikacije?

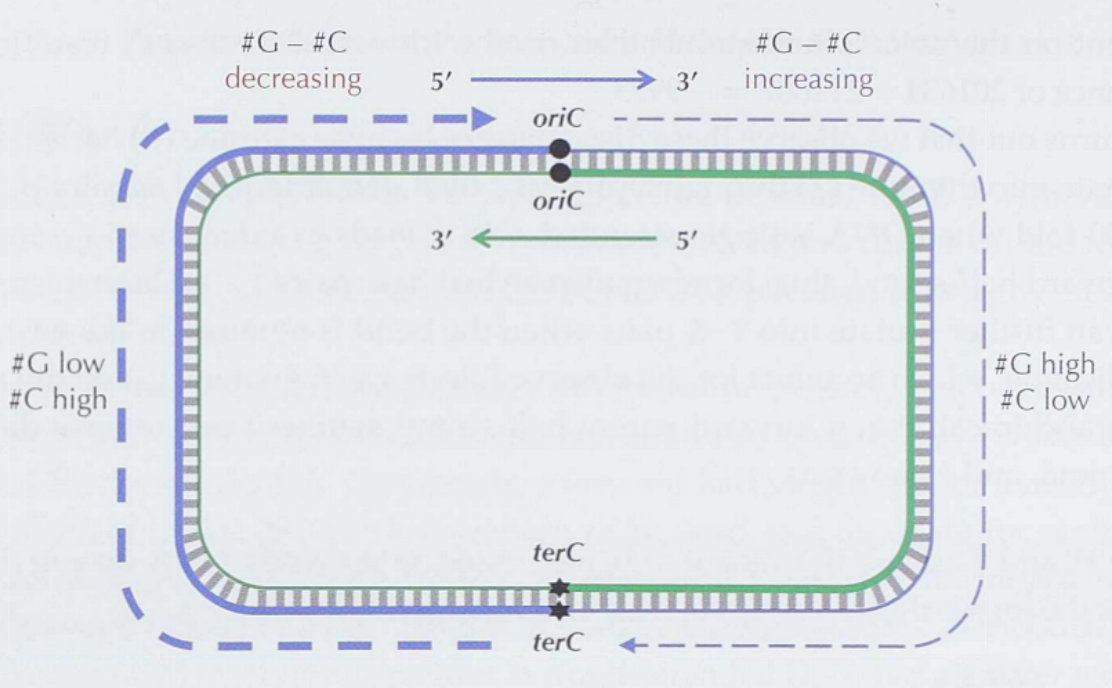
#### Iskrivljeni dijagrami

S obzirom na to da imamo veliku količinu statističkih podataka, pitamo se kako ih možemo upotrebiti da bismo došli do lokacije *oriC*-a? U tome nam mogu pomoći **iskrivljeni dijagrami**(engl. *skew diagram*). Osnovna ideja je da prođemo kroz genom i da računamo razliku između količine guanina(G) i citozina(C). Ako ova razlika raste, onda možemo pretpostaviti da se krećemo niz polulanac koji ide na desno (u nastavku samo polulanac, smer $5' -> 3'$), a ako razlika počne da se smanjuje, onda pretpostavljamo da smo na obrnutom polulancu ($3' -> 5'$). Zbog procesa koji se naziva deaminacija (gubljenje aminokiselina), svaki polulanac ima manjak citozina u poređenju sa guaninom, a svaki obrnuti polulanac ima manjak guanina u odnosu na citozin.

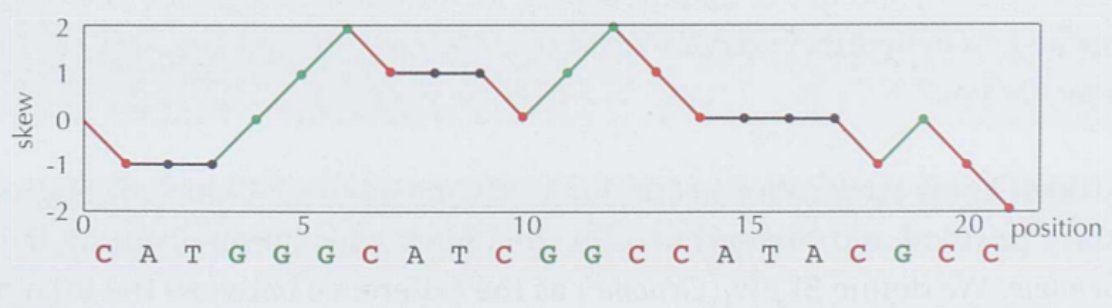
Posmatrajmo iskrivljeni dijagram bakterije Ešerihija Koli. Lako uočavamo minimalnu vrednost skew dijagrama.

Minimalna vrednost iz iskrivljenog dijagrama ukazuje baš na ovaj region:

Slika 1.11: Prikaz kretanja.




Slika 1.12: Iskrivljeni dijagram genoma Genome = CATGGGCATCGGCCATACGCC.

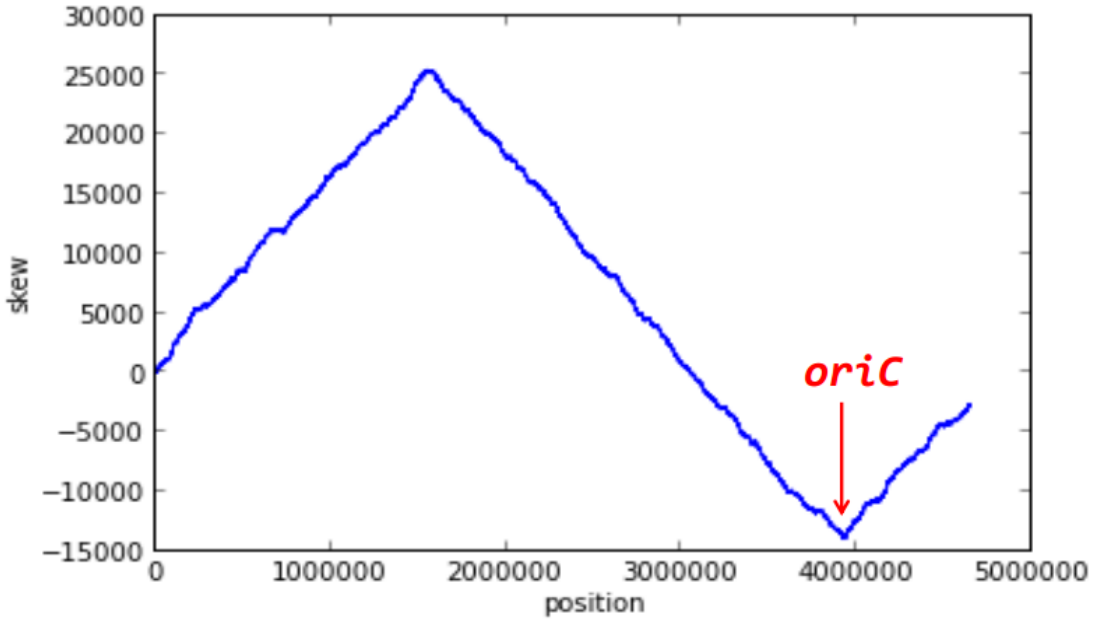



Slika 1.14: Region na koji pokazuje minimalna vrednost iskrivljenog dijagrama Ešerihije koli.

```
aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctcgcataacgcggta
tgaaaatggattgaagcccgggccgtggattctactcaactttgtcggcttgagaaagacc
tgggatcctgggtattaaaaagaagatctatttatttagagatctgttctattgtgatctc
ttattaggatcgcactgccctgtggataacaaggatccggcttttaagatcaacaacctgg
aaaggatcattaactgtgaatgatcggtgatcctggaccgtataagctgggatcagaatga
ggggttatacacaactcaaaaactgaacaacagttgttctttggataactaccggttgatc
caagcttcctgacagagttatccacagtagatcgcacgatctgtatacttatttgagtaaa
ttaacccacgatcccagccattcttctgccggatcttccggaatgtcgtgatcaagaatgt
tgatcttcagtg
```

Uočimo da u ovom regionu nema čestih 9-grama (koji se pojavljuju 3 ili više puta). Iz toga zaključujemo da, iako smo uspeli da nađemo *oriC* bakterije Ešerihija koli, nismo uspeli da nađemo

Slika 1.13: Iskrivljeni dijagram Ešerihije koli.




DNKA boksove. Međutim, pre nego što odustanemo od potrage, osmotrimo još jednom *oriC* Vibrio colerae, kako bismo pokušali da nađemo način da izmenimo naš algoritam i uspemo da lociramo DNKA boksove u Ešerihiji koli. Veoma brzo, može se uvideti da osim tri pojavljivanja ATGATCAAG i tri pojavljivanja CTTGATCAT, *oriC* Vibrio cholerae sadrži i dodatna pojavljivanja ATGATCAAC i CATGATCAT koji se razlikuju samo u jednom nukleotidu od gornjih niski. Ovo još više povećava šanse da smo naišli na prave DNKA boksove, a ima i biološkog smisla. Naime, DNKA se može vezati i za nesavršene DNKA boksove, one koji se razlikuju u nekoliko nukleotida.

Slika 1.15: Prikaz pojavljivanja nesavršenih niski nukleotida.

atca**ATGATCAAC**gtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaag**ATGATCAAG**agaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaag**CATGATCAT**ggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacct**CTTGATCAT**cgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagct**CTTGATCAT**gtt
tccttaaccctctatttttacggaaga**ATGATCAAG**ctgctgct**CTTGATCAT**cgtttc

Cilj nam je da sada izmenimo algoritam čestih reči (*FrequentWords*) tako da možemo da pronađemo DNKA boksove koji su predstavljeni čestim k-gramima, sa mogućim izmenama na pojedinim nukleotidima. Ovaj problem nazvaćemo problem čestih reči sa propustima.

**Problem čestih reči sa propustima.** Pronaći najčešće k-grame sa propustima u niski karaktera.
*Ulaz:* Niska Text i celi brojevi k i d.
*Izlaz:* Svi najčešći k-grami sa najviše d propusta u niski Text.

Pokušajmo, još jednom, sa pronalaskom DNKA boksova kod Ešerihije koli, tako što ćemo naći najčešće 9-grame sa propustima i komplementima u regionu *oriC* koji nam je predložen minimalnom vrednošću iskrivljenog dijagrama. Pokušaćemo sa malim prozorom koji ili počinje ili se završava ili je centriran na poziciji najmanje iskrivljenosti. Ovakvim izvođenjem pronalazimo TTATCCACA/TGTGGATAA kao najčešći 9-gram. Međutim, ovo nije jedini 9-gram. Za ostale 9-grame još uvek ne znamo čemu služe, ali znamo da nose skrivene informacije, da se grupišu unutar genoma i da većina njih nema veze sa replikacijom.

Slika 1.16: Prikaz pronađenih niski sa propustima i komplementima u *oriC* regionu Ešerihije koli.

aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctcgcataacgcgg
tatgaaaatggattgaagcccgggccgtggattctactcaactttgtcggcttgagaaa
gacctgggatcctgggtattaaaaagaagatctatttatttagagatctgttctattgt
gatctcttattaggatcgcactgcccTGTGGATAAcaaggatccggcttttaagatcaa
caacctggaaaggatcattaactgtgaatgatcggtgatcctggaccgtataagctggg
atcagaatgaggggTTATACACAactcaaaaactgaacaacagttgttcTTTGGATAAC
taccggttgatccaagcttcctgacagagTTATCCACAgtagatcgcacgatctgtata
cttatttgagtaaattaacccacgatcccagccattcttctgccggatcttccggaatg
tcgtgatcaagaatgttgatcttcagtg

## 1.3 Zadaci sa vežbi

U nastavku će biti predstavljeni zadaci sa vežbi na kursu rađeni u programskom jeziku Python.

### 1.3.1 FrequentWords

```
1  #frequent_words
2  def pattern_count(text, pattern):
3      count = 0
4      k = len(pattern)
5      for i in range(len(text) - k):
6          if text[i:i+k] == pattern:
7              count += 1
8      return count
9
10 def frequent_words(text, k, min_count):
11     frequent_patterns = set([])
12     count = []
13     n = len(text)-k
14     for i in range(n):
15         # Izvuci podnisku koji pocinje na $i$-toj poziciji i ima $k$ karaktera
16         pattern = text[i:i+k]
17         count.append(pattern_count(text, pattern))
18     max_count = max(count)
19     if max_count < min_count:
20         return []
21     for i in range(n):
22         if count[i] == max_count:
23             frequent_patterns.add(text[i:i+k])
24     return frequent_patterns
25
26 def main():
27     print(frequent_words('agctagatgctagctagctgatcgagctgatgcaggcagtgctagc', 4,
       ↪ 2))
28
29 if __name__ == "__main__":
30     main()
```

### 1.3.2 Faster FrequentWords

```
1  #faster frequent_words
2  def pattern_to_number(pattern):
3      if len(pattern) == 0:
4          return 0
5      last = pattern[-1]
6      prefix = pattern[:-1]
7      return 4 * pattern_to_number(prefix) + symbol_to_number(last)
8
9  def number_to_pattern(n, k):
10     if k == 1:
11         return number_to_symbol(n)
12     prefixIndex = n // 4 #celobrojno deljenje
13     r = n  %  4
```

```
14      symbol = number_to_symbol(r)
15      prefix  = number_to_pattern(prefixIndex, k-1)
16      return prefix + symbol
17
18  def symbol_to_number(c):
19      pairs = {
20          'a' : 0,
21          't' : 1,
22          'c' : 2,
23          'g' : 3
24      }
25      return pairs[c]
26
27  def number_to_symbol(n):
28      pairs = {
29          0 : 'a',
30          1 : 't',
31          2 : 'c',
32          3 : 'g'
33      }
34      return pairs[n]
35
36  def pattern_count(text, pattern):
37      count = 0
38      k = len(pattern)
39      for i in range(len(text) - k):
40          if text[i:i+k] == pattern:
41              count += 1
42      return count
43
44  def computing_frequencies(text,k):
45      frequency_array = [0 for i in range(4**k)] #4 ^ k
46      for i in range(len(text) - k):
47          pattern = text[i:i+k]
48          j = pattern_to_number(pattern)
49          frequency_array[j] += 1
50      return frequency_array
51
52  def faster_frequent_words(text, k, min_count):
53      frequent_patterns = set([])
54      frequency_array = computing_frequencies(text, k)
55      max_count = max(frequency_array)
56      if max_count < min_count:
57          return []
58      for i in range(4**k):
59          if frequency_array[i] == max_count:
60              pattern = number_to_pattern(i, k)
61              frequent_patterns.add(pattern)
62      return frequent_patterns
63
64  def main():
65      print(faster_frequent_words('agctagatgctagctagctgatcgagctgatgcaggcagtgctagc
    ↪ ', 4, 2))
```

```
66      #print(number_to_pattern(pattern_to_number('ta'),2))
67
68  if __name__ == "__main__":
69      main()
```

### 1.3.3 Skew Diagram

```
1   #GC-skew
2   import matplotlib.pyplot as plt
3
4   def draw_skew(skew):
5       x = [i for i in range(len(skew))]
6       ax = plt.subplot()
7       ax.plot(x, skew)
8       plt.show()
9
10  def calculate_skew(text):
11      skew = [0 for c in text]
12      last = 0
13      for i in range(0, len(text)):
14          if text[i] == 'g':
15              skew[i] = last + 1
16          elif text[i] == 'c':
17              skew[i] = last - 1
18          else:
19              skew[i] = last
20          last = skew[i]
21      return skew
22
23
24  def main():
25      text = "catgggcatcggccatacgcc"
26      print(calculate_skew(text))
27      draw_skew(calculate_skew(text))
28
29  if __name__ == "__main__":
30      main()
```

### 1.3.4 FrequentWords With Mismatches

```
1   # Prevodjenje nukleotida u brojeve
2   def symbol_to_number(c):
3       pairs = {
4           'a' : 0,
5           't' : 1,
6           'c' : 2,
7           'g' : 3
8       }
9
10      return pairs[c]
11
12  # Prevodjenje nukleotida u brojeve
13  def number_to_symbol(n):
14      pairs = {
```

```
15          0 : 'a',
16          1 : 't',
17          2 : 'c',
18          3 : 'g'
19      }
20
21      return pairs[n]
22
23  # Prevodjenje broja u odgovarajucu nukleotidnu sekvencu
24  def number_to_pattern(n, k):
25      if k == 1:
26          return number_to_symbol(n)
27
28      prefix_index = n // 4
29      r = n %  4
30      c = number_to_symbol(r)
31      prefix_pattern = number_to_pattern(prefix_index, k - 1)
32
33      return prefix_pattern + c
34
35
36
37  # Prevodjenje nukleotidne sekvence u odgovarajuci broj
38  def pattern_to_number(pattern):
39      if len(pattern) == 0:
40          return 0
41
42      last = pattern[-1:]
43      prefix = pattern[:-1]
44
45      return 4 * pattern_to_number(prefix) + symbol_to_number(last)
46
47
48
49  # Hamingova distanca, broj pozicija karaktera na kojima se tekstovi 1 i 2
        ↪ razlikuju,
50  # podrazumeva se da je duzina obe niske jednaka
51  def hamming_distance(text1, text2):
52      distance = 0
53
54      for i in range(len(text1)):
55          if text1[i] != text2[i]:
56              distance += 1
57
58      return distance
59
60
61
62  # Brojanje pojavljivanja podsekvenci u tekstu koje se od uzorka razlikuju na
        ↪ najvise d pozicija
63  def approximate_pattern_count(text, pattern, d):
64      count = 0
65
```

```
66      for i in range(len(text) - len(pattern)):
67          pattern_p = text[i:i+len(pattern)]
68
69          if hamming_distance(pattern, pattern_p) <= d:
70              count += 1
71
72      return count
73
74
75  # Pronalazenje svih niski susednih zadatom uzorku sa razlikama na najvise d
        ↪ pozicija
76  def neighbors(pattern, d):
77      if d == 0: # Ako je dozvoljena greska jednaka nuli onda je samo uzorak svoj
        ↪  sused bez gresaka
78          return set([pattern])
79
80      # Izlaz iz rekurzije:
81      if len(pattern) == 1: # Ako je duzina uzorka jednaka 1, a dozvoljena greska
        ↪  je veca od nule, onda je moguce iskoristiti bilo koji karakter
82          return set(['a', 't', 'c', 'g'])
83
84      neighborhood = set([])
85
86      suffix_neighbors = neighbors(pattern[1:], d) # Pronalaze se svi susedi
        ↪ duzine n-1
87
88      for text in suffix_neighbors:
89          if hamming_distance(pattern[1:], text) < d: # Ako se sused razlikuje na
        ↪  manje od d pozicija od podniske uzorka bez prvog karaktera
90              for x in ['a','t','c','g']:
91                  neighborhood.add(x + text) # Moguce je dodati bilo koji
        ↪ karakter na pocetak suseda i time dobiti najvise d razlika
92          else: # U suprotnom, sused se vec razlikuje na d pozicija od uzorka pa
        ↪ je dozvoljeno samo dodavanje ispravnog karaktera kako se
93              neighborhood.add(pattern[0] + text) # razlika ne bi povecala preko
        ↪ d
94
95      return neighborhood
96
97
98
99  def frequent_words_with_mismatches(text, k, d):
100     frequent_patterns = set([])
101
102     close = [0 for i in range(4**k)] # Kandidati za proveru
103     frequency_array = [0 for i in range(4**k)]
104
105     # Za svaki uzorak duzine k u tekstu evidentiraju se kandidat susedi cija
        ↪ pojavljivanja treba uzeti u razmatranje (niske koje se razlikuju od
        ↪ uzorka na najvise d pozicija)
106     for i in range(len(text) - k):
107         neighborhood = neighbors(text[i:i+k], d) # Pronalaze se kandidati
108
```

```
109         for pattern in neighborhood:
110             index = pattern_to_number(pattern) # Svaka kandidat sekvenca se
    ↪ prevodi u svoj odgovarajuci indeks
111             close[index] = 1 # i evidentira
112 
113     # Za svaku kandidat sekvencu (koja je do sada pronadjena u tekstu sa
    ↪ greskom d), broji se pojavljivanje njenih d-suseda
114     for i in range(4**k):
115         if close[i] == 1:
116                 pattern = number_to_pattern(i, k)
117                 frequency_array[i] = approximate_pattern_count(text, pattern, d
    ↪ )
118 
119     max_count = max(frequency_array)
120 
121     # Pronalaze se one sekvence duzine k cija su pojavljivanja najzastupljenija
122     for i in range(4**k):
123         if frequency_array[i] == max_count:
124             pattern = number_to_pattern(i, k)
125             frequent_patterns.add(pattern)
126 
127     return frequent_patterns
128 
129 
130 
131 
132 def main():
133     print(frequent_words_with_mismatches('
    ↪ tgactatcatcgtagtatcgatgtgcacacacgtgcgcgcgcgcgccctgtacatgatc', 5, 2))
134 
135 if __name__ == "__main__":
136     main()
```

# Glava 2

# Kako složiti genomsku slagalicu od milion delova?

Kao i do sada, iz naslova ovog poglavlja nam verovatno nije jasno o kakvom problemu je reč. U ovom poglavlju predstavićemo dati problem i prikazati kako možemo primeniti grafovske algoritme nad problemom slaganja genomske slagalice. Poglavlje ćemo započeti pričom o sekvenciranju genoma. Do sada smo videli šta za nas znači pojam DNK – da se podsetimo, to je niska karaktera nad azbukom $\Sigma = \{A, C, G, T\}$. Biološki posmatrano, DNK je molekul koji se nalazi u svakoj ćeliji svakog organizma i da je u njemu zapisan način pravilnog razvoja i funkcionisanja svakog organizma.

## 2.1 Šta je sekvenciranje genoma?

Sa biološke strane, genom jednog organizma predstavlja njegov genetski materijal. Pod genetskim materijalom smatramo materijal koji se nasleđuje i koji svi predstavnici jedne vrste dele u velikoj meri. Kod većine organizama, genetski materijal je sadržan u DNK, odnosno, genom je ekvivalentan sa DNK molekulima. Kod nekih drugih organizama koji su u manjini, kao što su, na primer, virusi, važi da oni ne sadrže DNK i njihov genetski materijal se nalazi u ribonukleinskoj kiselini – RNK – o kojoj će biti reči u nastavku.

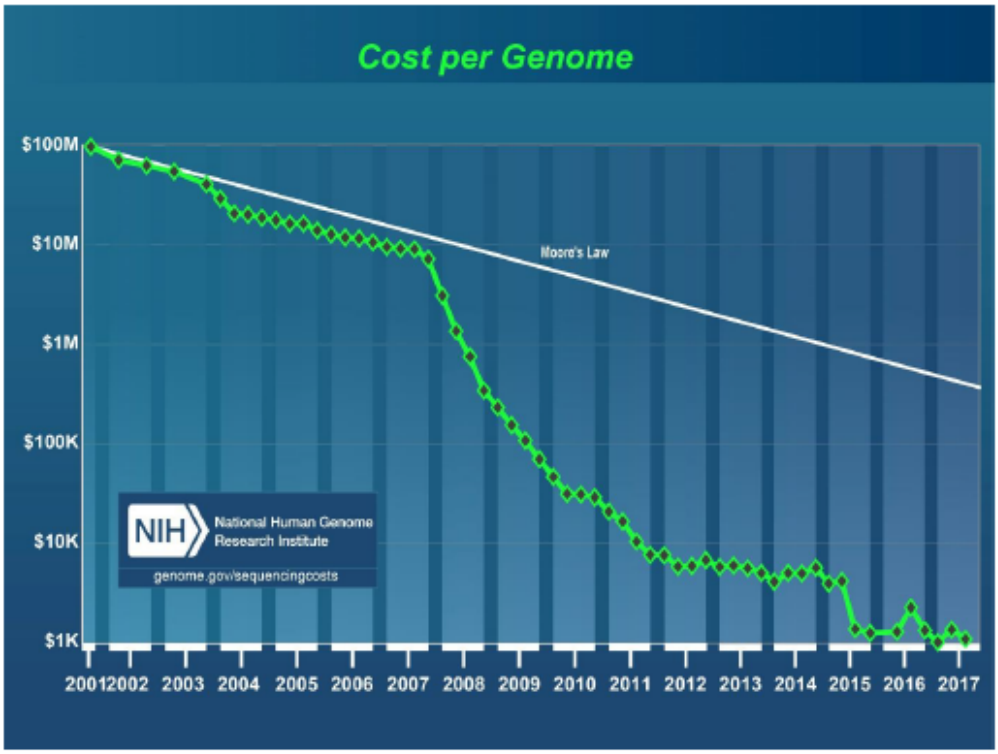
Kod čoveka, genom sadrži oko tri milijarde nukleotida. Dakle, sa računarske strane posmatrano genom je niska karaktera nad azbukom $\Sigma = \{A, C, G, T\}$. Kompleksnost organizma nije u relaciji sa veličinom genoma. Genomi nekih organizama su i stotinu puta veći od humanog genoma. Na primer, jedna vrsta amebe ima 670 milijardi nukleotida ili jedna vrsta kaktusa koja raste u Japanu ima 150 milijardi nukleotida.

Sekvenciranje genoma podrazumeva otkrivanje izgleda genoma. U pitanju je eksperimentalan proces – da bismo saznali šta se nalazi u sastavu jednog genoma, potreban nam je uzorak tkiva odgovarajuće vrste. U nastavku dajemo kratak pregled razvoja sekvenciranja genoma.

### 2.1.1 Kratka istorija sekvenciranja genoma

Kao što smo videli, sekvenciranje genoma je eksperimentalan proces, za koji je neophodna veoma ozbiljna tehnologija. Pre svega možemo govoriti o razvoju fizičko-hemijskih tehnika koje bi dovele do mogućnosti saznavanja sastava genoma. Walter Gilbert i Frederick Sanger su 1977. godine razvili nezavisne metode sa sekvenciranje genoma, za koje su, 1980. godine, podelili Nobelovu nagradu. Međutim, iako su njihovi metodi bili pionirski u ovoj oblasti, njihove metode za sekvenciranje su bile veoma skupe – za sekvenciranje humanog genoma je bilo potrebno 3 milijarde dolara.

Krajem 2000-ih Sanger metodom je sekvencioniran veliki broj genoma. Visoka cena je bila ograničavajući faktor i za dalji napredak je bila neophodna nova tehnologija sekvencioniranja. *NGS* (skr. *Net Generation Sequencing*) predstavlja metode nove generacije sekvencioniranja, odnosno, novu generaciju mašina sekvencera koji vrše sekvencioniranje. *Illumina*, jedan od proizvođača sekvencera, smanjuje trošak sekvencioniranja humanog genoma sa 3 milijarde na 10 hiljada dolara. Kompanija *Complete Genomics* otvara genomsku fabriku u Silikonskoj dolini koja sekvencionira stotine genoma mesečno. Pekinški genomski institut (*Beijing Genome Institute*, skr. *BGI*) preuzima Complete Genomics 2013. godine i postaje najveći svetski centar za sekvenciranje genoma. Na slici **??** prikazano je kako se cena sekvencioniranja menjala godinama.




Slika 2.1: Cena sekvencioniranja kroz istoriju.

### 2.1.2 Sekvenciranje ličnih genoma

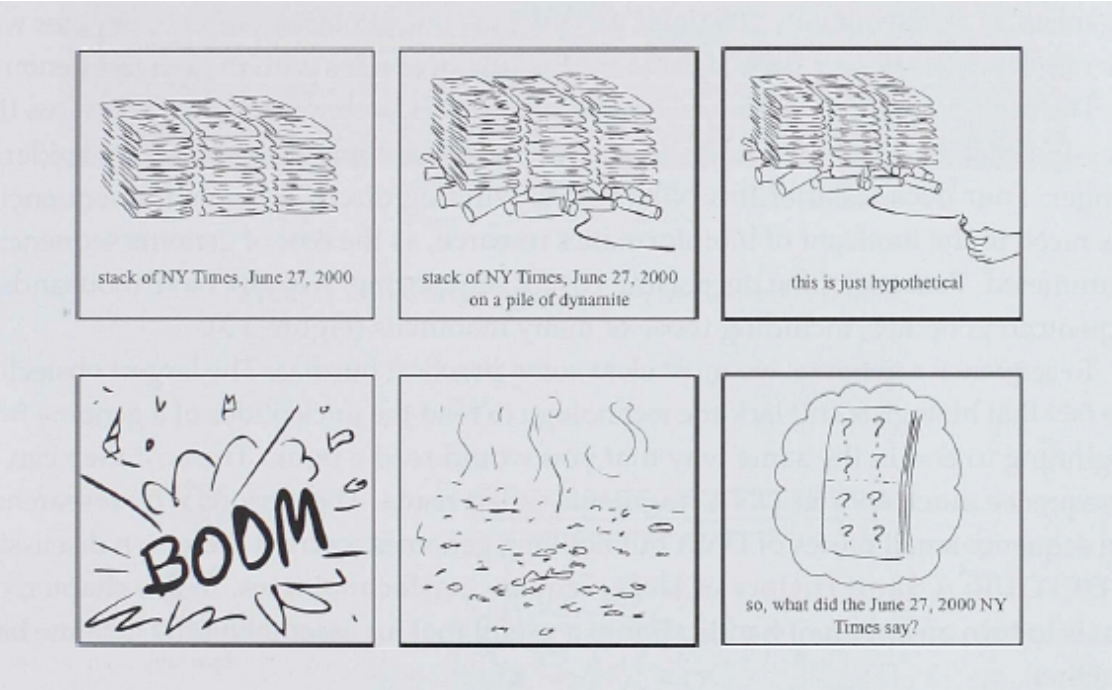
Prirodno je zapitati se o značaju poznavanja sastava genoma. Što je tiče biljaka, neke od primena sekvenciranja su: razvoj novih biljnih vrsta u poljoprivredi, određivanje pogodnog podneblja za neku biljnu vrstu, u farmaciji, i dr. Međutim, najznačajnija primena je u sekvenciranju ličnih genoma.

Genomi se kod različitih ljudi razlikuju na malom broju pozicija (u proseku sadrže jednu mutaciju na hiljadu nukleotida). Ova razlika je odgovorna za, na primer, različite visine kod ljudi, da li će imati sklonost ka visokom holesterolu ili ne, za veliki broj genetskih bolesti, itd.

Godine 2010. Nicholas Volker je postao prvo ljudsko biće čiji je život spašen zahvaljujući genomskom sekvencioniranju. Lekari nisu mogli da postave tačnu dijagnozu i morali su da ga podvrgnu velikom broju operacija pokušavajući da je utvrde. Sekvenciranje je otkrilo retku mutaciju na jednom genu (XIAP) koja je bila povezana sa oštećenjem njegovog imunog sistema. Ovo otkriće je navelo lekare na adekvatnu terapiju koja je rešila problem.

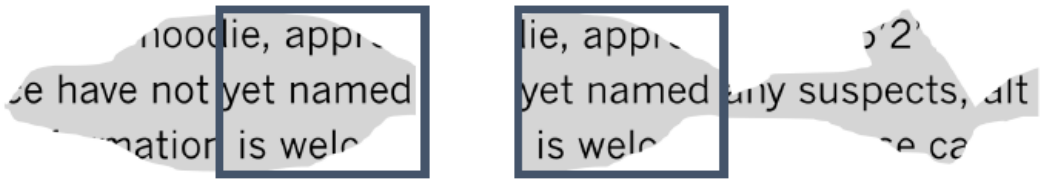
## 2.2 Eksplozija u štampariji

Razmotrimo sledeći primer. Neka imamo hiljadu kopija istog izdanja novina na jednoj gomili, a ispod njih postavljen je dinamit. Upalimo fitilj i zamislimo da nije sve samo izgorelo već da se raspršilo u milione delića papira. Kako možemo da iskoristimo te deliće da bismo saznali koje su bile vesti iz tog izdanja? Ovaj problem nazvaćemo *Problem novina* (videti sliku **??**).




Slika 2.2: Problem novina poslužiće nam u razumevanju problema slaganja genoma.

Problem novina je mnogo teži nego što izgleda. Kako smo imali više kopija istog izdanja, i kako smo izgubili neki deo informacija prilikom eksplozije, ne možemo samo da prilepimo deliće novina kao da su slagalica. Umesto toga, potrebno je da preklopimo delove različitih novina kako bismo rekonstruisali jedan primerak.



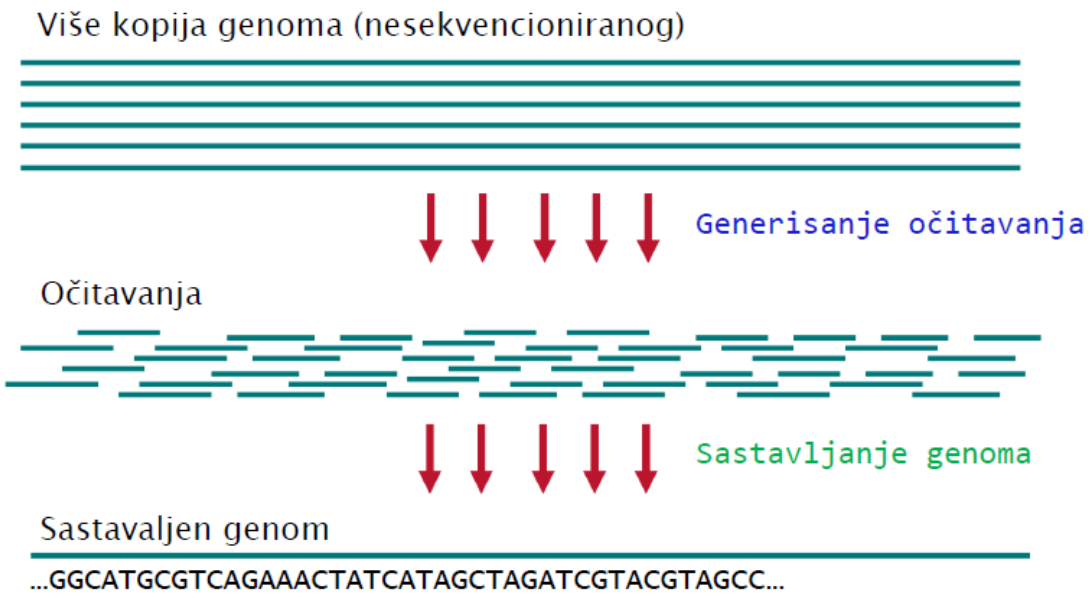
Slika 2.3: Spajanje delova različitih novina koji se jednim delom preklapaju.

Određivanje redosleda nukleotida u genomu, odnosno sekvenciranje genoma, predstavlja bitan problem u bioinformatici. Već smo pomenuli da dužine genoma variraju – humani genom je dugačak oko 3 milijarde nukleotida, dok je genom jednoćelijskog organizma *Amoeba dubia* čak 200 puta duži.

Razmotrimo sada povezanost problema novina i sekvenciranja genoma. Kopije izdanja u problemu novina odgovaraju ono predstavlja ulaz u sekvencerima – uzorak tkiva. Moderne

mašine za sekvenciranje ne mogu da pročitaju ceo genom nukleotid po nukleotid od početka do kraja (kao što bismo pročitali knjigu). Mogu samo da iseckaju genom i generišu njegova kratka *očitavanja* (engl. *reads*). Kako to zapravo funkcioniše?

Na slici **??** ilustrovan je proces sekvenciranja. Sekvencer dobija milione kopija istog genoma. Zatim vrši očitavanja čime dobijamo deliće odnosno kratke podniske. Neki delovi odnosno očitavanja biće izgubljena (kao delići novina u eksploziji, dakle gubimo deo informacija). Očitavanja su izmešana i ono što nam sekvencer daje je zapravo kolekcija podniski koje treba spojiti u jednu. Sastavljanje genoma nije isto kao i slaganje slagalice – moramo da koristimo preklapajuća očitavanja da bismo rekonstruisali genom.




Slika 2.4: Ilustracija problema.

## 2.3 Problem sekvenciranja genoma

Do sada smo videli šta predstavlja sekvenciranje genoma, koja je njegova biološka podloga i kako se on definiše kao biološki problem. Pređimo sada na formulisanje računarskog problema sekvenciranja genoma kao problem rekonstrukcije niske.

**Problem 1** (Problem sekvencioniranja genoma)**.** *Rekonstruisati genom na osnovu očitavanja.*
*Ulaz: Kolekcija niski* Reads*.*
*Izlaz: Niska* Genome *rekonstruisana na osnovu* Reads*.*

Ovo nije dobro definisan problem. Potrebno je uvesti dodatne pojmove kako bismo uspeli da problem sekvencioniranja genoma predstavimo kao problem rekonstrukcije niske.

Definišemo pojam *k-gramski sastav niske* na sledeći način. $k$-gramski sastav niske $Text$, u oznaci $Composition_k(Text)$, predstavlja kolekciju podniski dužine $k$ niske $Text$, pri čemu su u kolekciju uključeni duplikati. Na primer, neka je $Text = TAATGCCATGGGATGTT$. Njen 3-gramski sastav niske $Text$ izgleda:

```
Composition_3(TAATGCCATGGGATGTT) =
               TAA
                AAT
                 ATG
                  TGC
                   GCC
                    CCA
                     CAT
                      ATG
                       TGG
                        GGG
                         GGA
                          GAT
                           ATG
                            TGT
                             GTT
```

odnosno, ako kolekciju uredimo po leksikografskom poređenju:

```
Composition_3(TAATGCCATGGGATGTT) =
AAT ATG ATG ATG CAT CCA GAT GCC GGA GGG GTT TAA TGC TGG TGT
```

Sada možemo malo bolje da definišemo problem.

**Problem 2** (Problem rekonstrukcije niske)**.** *Rekonstruisati nisku na osnovu njenog k-gramskog sastava.*
*Ulaz: Kolekcija k-grama.*
*Izlaz: Niska* Genome *takva da je* $Composition_k(Genome)$ *ekvivalentno kolekciji k-grama.*

Započnimo prvo sa naivnim pristupom rešavanju ovog problema. Odaberimo jedan $k$-gram za početni. Zatim nižemo ostale tako da se sufiks poslednjeg odabranog poklopi sa prefiksom nekog od preostalih $k$-grama. Pri tome, ako ima više takvih $k$-grama, biramo proizvoljan jedan. Na ovaj način možemo doći do rešenja, ali je veoma skupo. Pri tome, velika je šansa da ćemo se negde zaglaviti (tj. nijedan od preostalih $k$-grama neće biti kandidat za nadovezivanje na tekuću nisku) ili zbog izbora početnog $k$-grama ili zbog izbora nekog od preostalih $k$-grama kada je postojalo više odgovarajućih. Sledeći primer ilustruje ovaj problem.

Neka nam je dat sledeći 3-gramski sastav: `AAT ATG ATG ATG CAT CCA GAT GCC GGA GGG GTT TAA TGC TGG TGT`. Treba rekonstruisati nisku koja ima takav sastav. Biramo početni 3-gram, neka to bude na primer `TAA`. Zatim na njega treba nadovezati 3-gram koji počinje njegovim sufiksom dužine 2, odnosno onaj 3-gram koji ima prefiks `AA`. U našem slučaju, postoji jedan takav 3-gram i njega nadovezujemo na tekuću nisku, tako da sada imamo `TAAT`. Zatim biramo 3-gram čiji je prefiks `AT`. Ovog puta imamo 3 kandidata, ali, na našu sreću, sva tri su isti 3-grami, `ATG`. U takvom slučaju nije bitno koji smo odabrali, jer su svi jednaki. Nadovezujemo ga na tekuću nisku i dobijamo `TAATG`. Tražimo 3-grame sa prefiksom `TG`, koji do sad nisu upotrebljeni. Ponovo pronalazimo 3 kandidata. Međutim, u ovom slučaju, svi kandidati predstavljaju različite 3-grame, a to su `TGC`, `TGG` i `TGT`. Naivni pristup kaže da biramo jedan od njih, i recimo da smo odabrali `TGT` i dobili nisku `TAATGT`. Sada nam je potreban 3-gram sa prefiksom `GT` i tu dolazi do zaglavljivanja. Imamo još 3-grama koji nisu iskorišćeni za rekonstrukciju niske, ali nijedan ne možemo da iskoristimo u ovom trenutku. U takvim situacijama treba se vratiti u nazad do koraka u kom je bilo više kandidata.

## 2.4 Rekonstrukcija niske kao problem Hamiltonove putanje

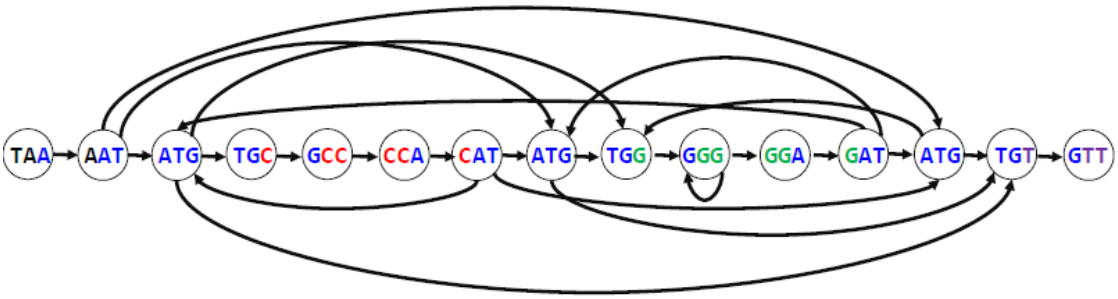
Videli smo da nam naivni pristup ne odgovara i moramo smisliti bolje rešenje. Mogli bismo da iskoristimo znanja iz teorije grafova za rešavanje ovakvog problema. U tom slučaju, prvi zadatak je da našu nisku predstavimo u vidu grafa.

### 2.4.1 Genom kao putanja

Vratimo se na prethodni primer. Dat nam je naredni 3-gramski sastav:

```
1  Composition_3(TAATGCCATGGGATGTT) =
2                TAA
3                 AAT
4                  ATG
5                   TGC
6                    GCC
7                     CCA
8                      CAT
9                       ATG
10                       TGG
11                        GGG
12                         GGA
13                          GAT
14                           ATG
15                            TGT
16                             GTT
```

Ovakav 3-gramski sastav možemo predstaviti kao graf na sledeći način. Svakom čvoru u grafu odgovara jedan od $k$-grama. Zatim, potrebne su nam grane koje će povezati te čvorove. Dva čvora su povezana usmerenom granom ako izlazni čvor ima sufiks koji je jednak prefiksu ulaznog čvora te grane, kao što je prikazano na slici **??**.




Slika 2.5: Graf koji odgovara 3-gramskom sastavu niske $TAATGCCATGGGATGTT$.

Jasno je da postoji više puteva u ovom grafu. Postavlja se pitanje – da li možemo da pronađemo genomsku putanju u ovom grafu, od svih koje postoje?

Podsetimo se šta je Hamiltonova putanja. Hamiltonova putanja je putanja koja posećuje svaki čvor u grafu tačno jednom. To je upravo ono što nam je potrebno za rešavanje problema. Svaki čvor predstavlja jedan $k$-gram i potrebno nam je da svi $k$-grami budu uključeni u rekonstruisanu nisku tačno jednom.

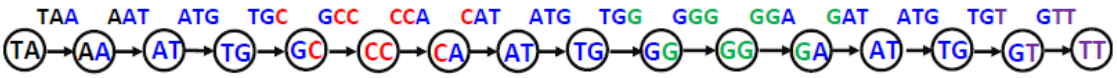
**Problem 3** (Problem Hamiltonove putanje)**.** *Naći Hamiltonovu putanju u grafu.*
*Ulaz: Graf.*
*Izlaz: Putanja koja posećuje svaki čvor u grafu tačno jednom.*

Iako deluje kao da smo rešili sve probleme, zapravo smo naišli na još jednu veliku prepreku. Naime, pronalaženje Hamiltonovog puta u grafu je NP-kompletan problem, što znači da ne postoji efikasan algoritam koji to radi. U tom slučaju, moramo da se vratimo na početak, a to je predstavljanje $k$-gramskog sastava grafom.

## 2.5 Rekonstrukcija niske kao Ojlerove putanje

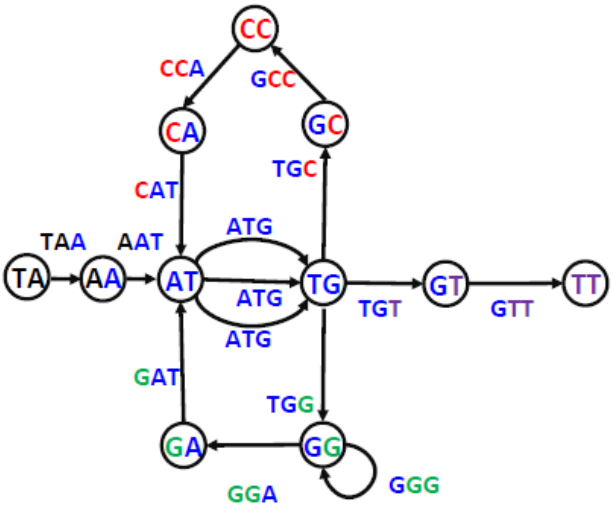
U prethodnoj sekciji, $k$-grame smo predstavili čvorovima u grafu i u njemu tražili Hamiltonov put, odnosno, put koji obilazi svaki čvor tačno jednom. Videli smo da za taj problem još uvek nije poznat efikasan algoritam pa se sada pitamo kako možemo izmeniti graf tako da ne zahteva traženje Hamiltonove putanje.

Ono što se javlja kao ideja jeste obeležavanje grana umesto čvorova. Dakle, svaka grana biće obeležena jednim $k$-gramom. Izlazni čvor biće obeležen prefiksom $k$-grama te grane, dok će ulazni čvor biti obeležen sufiksom istog tog $k$-grama. Slika **??** ilustruje ovaj postupak za nisku $TAATGCCATGGGATGTT$.




Slika 2.6: Graf koji odgovara 3-gramskom sastavu niske $TAATGCCATGGGATGTT$. Grane su obeležene 3-gramima, a čvorovi 2-gramima koji predstavljaju prefikse i sufikse.

Primećujemo da su neki čvorovi obleženi identično (na primer, imamo tri čvora sa oznakom $AT$). Sve čvorove koji imaju istu oznaku treba spojiti u jedan, pri čemu zadržavamo sve grane koje su ulazile u taj čvor ili su izlazile iz njega. Ponavljamo postupak dokle god imamo čvorove koji imaju istu oznaku i na kraju dobijamo graf koji nazivamo *De Brojnov graf*. Slika **??** ilustruje De Brojnov graf dobijen ovom procedurom od polaznog grafa sa slike **??**.




Slika 2.7: De Brojnov graf koji odgovara niski $TAATGCCATGGGATGTT$.

Ovime smo dobili novu reprezentaciju niske pomoću grafa. Prirodno se postavlja naredno pitanje – gde se nalazi niska *Genome* u ovoj reprezentaciji grafa? Kako nam se 3-grami sada nalaze na granama, a ne u čvorovima, potrebno je da pronađemo putanju u grafu koja prolazi sve grane tačno jednom. Takav put nazivamo *Ojlerova putanja*. Srećom, algoritam za pronalaženje Ojlerove putanje u grafu nije NP-kompletan i možemo efikasno da je pronađemo.

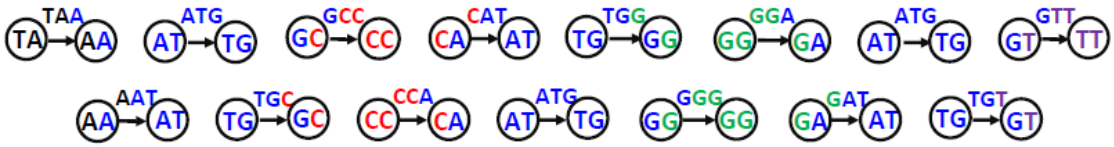
**Problem 4** (Problem Ojlerove putanje)**.** *Pronaći Ojlerovu putanju u grafu.*
*Ulaz: Graf.*
*Izlaz: Putanja koja posećuje svaku granu u grafu tačno jednom.*

Sada znamo kako možemo da dobijemo nisku kada znamo De Brojnov graf koji odgovara njenom $k$-gramskom sastavu. Međutim, konstruisali smo De Brojnov graf na osnovu genoma, ali u realnim primenama, genom je nepoznat.

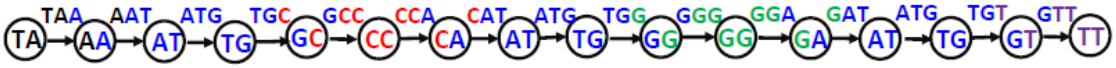
## 2.6 De Brojnovi grafovi na osnovu kolekcije $k$-grama

Videli smo kako možemo od zadate niske pronaći De Brojnov graf. Nažalost, u primenama nije nam poznata niska, ali znamo njen $k$-gramski sastav. Postavlja se pitanje kako možemo konstruisati De Brojnov graf od $k$-gramskog sastava niske.

Za svaki $k$-gram pravimo dva čvora i jednu granu – oznaka grane je upravo taj $k$-gram, oznaka izlaznog čvora je prefiks, a ulaznog čvora je sufiks datog $k$-grama. Time dobijamo nepovezani graf kao na slici **??**. Zatim lepimo identične čvorove sve dok ne dobijemo graf čiji svi čvorovi imaju različite oznake. Na slici **??** dat je jedan korak ovog postupka, međutim, tu nije kraj jer i dalje postoje čvorovi sa istim oznakama.




Slika 2.8: Svaki k-gram prestavljen je pomoću dva čvora i jedne grane.




Slika 2.9: Prvi korak u postupku lepljenja čvorova. Napomenimo da postupak nije završen.

Po završetku postupka dobijamo de Brojnov graf koji je isti kao onaj koji smo dobili kada smo znali nisku, odnosno, graf na slici **??**. Svaka grana je označena jednim $k$-gramom, a svaki čvor je označen prefiksom, odnosno, sufiksom odgovarajuće izlazne, odnosno, ulazne grane, redom. Naravno, čvovori koji imaju identične oznake su zalepljeni.

## 2.7 Ojlerova teorema

Za rešavanje problema Ojlerove putanje koji smo predstavili u prethodnoj sekciji možemo iskoristiti rešenje narednog problema. Ovaj problem je značajan jer postoji teorema koja ga prati, a koja određuje uslove za njegovo rešavanje.

**Problem 5** (Problem Ojlerovog ciklusa)**.** *Pronaći ciklus u Ojlerovom grafu.*
*Ulaz: Graf.*
*Izlaz: Ciklus koji posećuje svaku granu u grafu tačno jednom.*

Kažemo da je graf Ojlerov ako sadrži Ojlerov ciklus. Ispostavlja se da postoje određene karakteristike koje određuju da li je graf Ojlerov. Uvedimo pojmove povezan graf i balansiran graf. Kažemo da je graf *povezan* ako za ma koja dva čvora postoji putanja koja ih povezuje. Graf je *balansiran* ako za svaki čvor važi da mu je izlazni stepen jednak ulaznom. Naredna teorema govori o potrebnim i dovoljnim uslovima da graf bude Ojlerov.

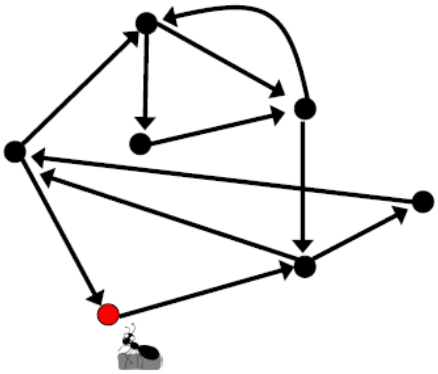
**Teorema 2.1** (Ojlerova teorema)**.** *Svaki Ojlerov graf je balansiran. Svaki povezan graf i balansiran graf je Ojlerov.*

### 2.7.1 Dokaz Ojlerove teoreme

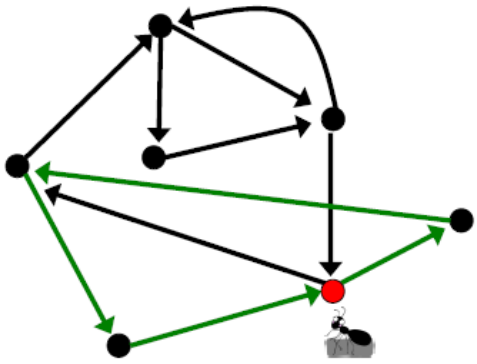
Neka nam je dat povezan balansiran graf. Da bismo pokazali da graf sadrži Ojlerov ciklus, postavićemo mrava na bilo koji od čvorova tog grafa, kao na slici **??**. Zašto baš mrav? Poznato je da mravi nikada ne idu istim putem dva puta pa smo sigurni da će naš mrav proći svaku granu tačno jednom.

Puštamo mrava da slučajno odabira grane kojima će se kretati. Ako je veoma pametan, obići će svaku granu jednom i vratiće se u početni čvor. Međutim, velike su šanse da nije veoma pametan i da će se u nekom čvoru zaglaviti, odnosno, neće imati granu koju već nije obišao.

Da li mrav može da se zaglavi u bilo kom čvoru? Ispostavlja se da može da se zaglavi samo u početnom čvoru (jer je graf balansiran). U trenutku kada se zaglavio on je napravio ciklus. Samo, taj ciklus nije Ojlerov jer još uvek nije obišao sve grane. Ideja je da odabere drugačiji početni čvor iz kog će krenuti obilazak. Koji čvor će izabrati? Treba da izabere čvor iz ciklusa koji ima izlaznih grana koje još uvek nije obišao (slika **??**).



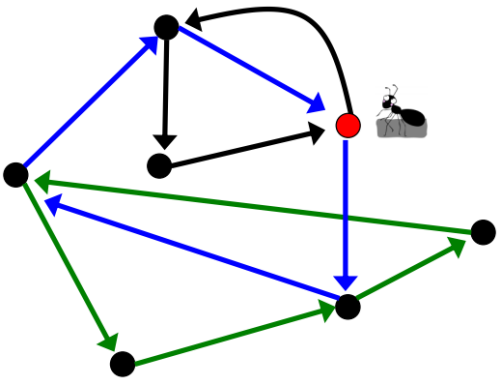
Slika 2.10: Mrav je postavljen u crveni čvor i odatle kreće obilazak povezanog balansiranog grafa.



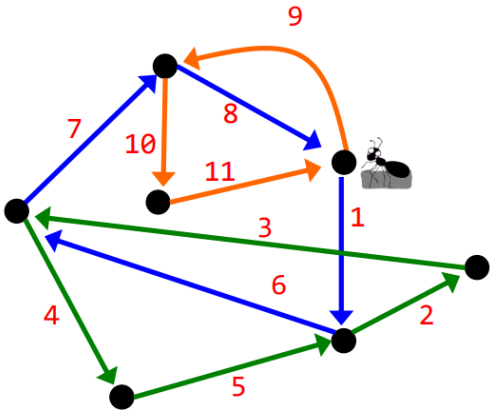
Slika 2.11: Mrav se zaglavio i pokušava ponovo iz drugog čvora koji pripada ciklusu i ima izlazne grane koje nisu posećene.

Sada, mrav pokušava ispočetka iz novog čvora. Prvo obilazi ciklus koji je već pronašao u prethodnom pokušaju, a zatim nastavlja obilazak preko neposećenih grana. Na taj način, ciklus se uvećava dok se ne dođe do Ojlerovog. Ukoliko se ponovo zaglavi (slika **??**) ponovo bira novi početni čvor, obilazi pronađeni ciklus (koji i dalje nije Ojlerov) i tako sve dok ne uspe da obiđe sve grane (slika **??**).

Dokaz Ojlerove teoreme daje primer konstruktivnog dokaza, koji ne dokazuje samo željeni rezultat, već pruža metod za konstrukciju onoga što nam je potrebno. Ukratko, pratili smo kretanje mrava dok nije pronašao Ojlerov ciklus u povezanom balansiranom grafu, što je sumirano u algoritmu `EulerianCycle`.



Slika 2.12: Mrav se ponovo zaglavio i pokušava od novog čvora.




Slika 2.13: Mrav je konačno uspeo da pronađe dobar početni čvor i Ojlerov ciklus. Grane su obeležene redosledom kojim su posećene.

```
1  EulerianCycle(BalancedGraph)
2  begin
3      form a Cycle by randomly walking in BalancedGraph (avoiding already visited
       ↪ edges)
4      while Cycle is not Eulerian
5          select a node newStart in Cycle with still unexplored outgoing edges
6          form a Cycle' by traversing Cycle from newStart and randomly walking
       ↪ afterwards
7          Cycle ← Cycle'
8      return Cycle
9  end
```

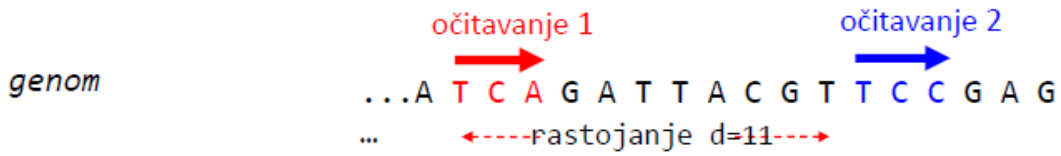
Ovaj algoritam radi u linearnom vremenu. Da bi se zaista postigla ta efikasnost, potrebne su efikasne struktutre podataka za održavanje ciklusa koje mrav pronalazi kao i za liste neiskorišćenih grana za svaki čvor i lista čvorova u trenutnom ciklusu koji imaju neiskorišćene grane.

## 2.8 Sastavljanje parova očitavanja

Deluje kao da su svi naši problemi rešeni. Međutim, može se javiti više Ojlerovih putanja u grafu. Srećom, i za ovo imamo jednostavno rešenje.

### 2.8.1 DNK sekvenciranje sa parovima očitavanja

Imamo više identičnih kopija genoma i na slučajnim pozicijama sečemo genom na fragmente iste dužine *InsertLength*. Zatim generišemo *parove očitavanja* – dva očitavanja sa krajeva svakog fragmenta na jednakoj, fiksiranoj udaljenosti. Pod *uparenim k-gramom* podrazumevamo par $k$-grama na fiksiranom rastojanju $d$ u genomu. *Upareni k-gramski sastav*, u oznaci $PairedComposition_k(Text)$, sastoji se od svih $k$-grama niske $Text$ i njihovih parova.




Slika 2.14: TCA i TCC na rastojanju $d = 11$ čine jedan upareni 3-gram.

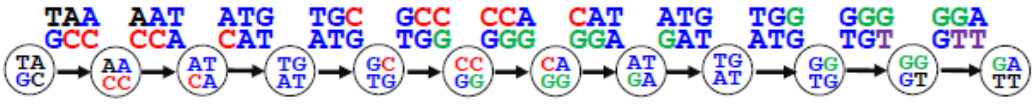
Dajmo jedan primer. Neka imamo nisku TAATGCCATGGGATGTT, i upareni 3-gram TAA i GCC. Upareni $k$-gramski sastav date niske prikazan je na slici **??**.


Slika 2.15: *PairedComposition* niske TAATGCCATGGGATGTT i njegov leksikografski poredak.

Sada možemo formulisati naredni problem.

**Problem 6** (Problem rekonstrukcije niske na osnovu parova očitavanja)**.** *Rekonstruisati nisku na osnovu njenih uparenih $k$-grama.*
*Ulaz: Kolekcija uparenih $k$-grama.*
*Izlaz: Niska $Text$ takva da je $PairedComposition_k(Text)$ jednak kolekciji uparenih $k$-grama.*

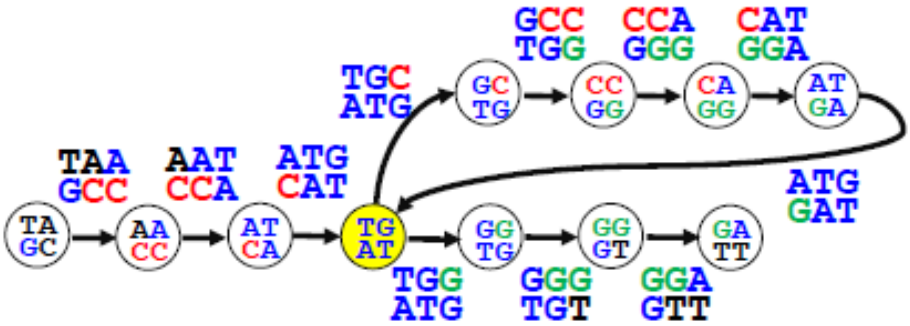
Kako konstruisati upareni De Brojnov graf na osnovu uparenog $k$-gramskog sastava? Postupak je sličan prethodnom slučaju, kada nismo imali parove. Pretpostavimo da je dat genom (niska *Genome*). Posmatrajmo genom kao putanju u grafu obeleženom na osnovu njegovog uparenog $k$-gramskog sastava (videti sliku **??**). Svaka grana obeležena je uparenim $k$-gramom, a svaki čvor uparenim prefiksom, odnosno, sufiksom $k$-grama.



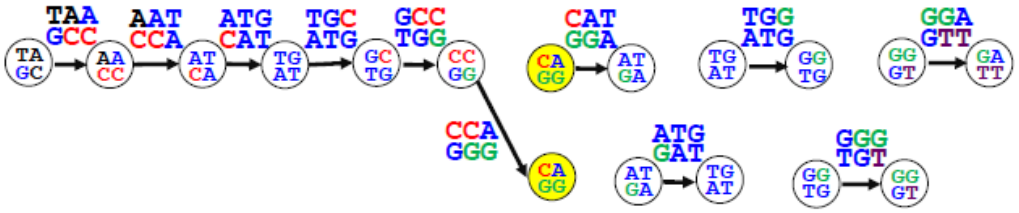

Slika 2.16: Graf koji odgovara uparenom 3-gramskom sastavu niske TAATGCCATGGGATGTT.

Potrebno je zalepiti čvorove sa istom oznakom, tako da svi čvorovi budu jedinstveno obeleženi. Postupak je identičan prethodnom slučaju, odnosno, kada nismo imali parove. Primetimo da sada imamo mnogo manje lepljenja jer imamo samo dva čvora sa istom oznakom (TG AT), (videti sliku **??**).

Kao i u prethodnom slučaju, pretpostavili smo da je dat genom (niska *Genome*), što često nije slučaj. Posmatrali smo genom kao putanju u grafu obeleženom na osnovu njegovog uparenog $k$-gramskog sastava. Sada pretpostavimo da nije dat genom već samo upareni $k$-gramski sastav. Za svaki upareni $k$-gram pravimo dva čvora i jednu granu, zatim lepimo identične čvorove (videti **??**), i na kraju dobijamo upareni De Brojnov graf, kao onaj na slici **??**.




Slika 2.17: Dva čvora sa oznakom (`TG AT`) spajaju se u jedan pri čemu su sve grane, incidentne sa tim čvorovima, očuvane.



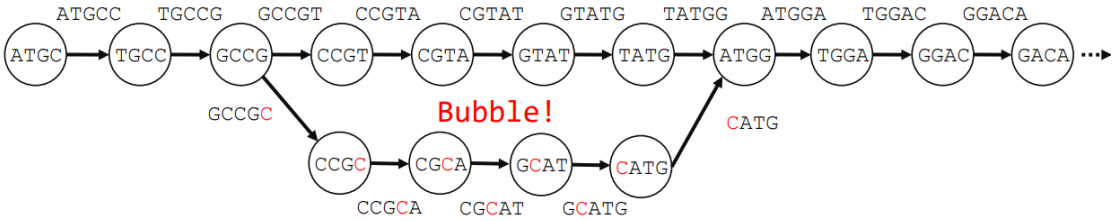

Slika 2.18: Konstrukcija uparenog De Brojnovog grafa na osnovu uparenih $k$-grama.

Dakle, upareni De Brojnov graf, na osnovu kolekcije uparenih $k$-grama, dobijamo tako što svaku granu označavamo jednim uparenim $k$-gramom. Zatim, svaki čvor označavamo prefiksima, odnosno, sufiksima odgovarajuće izlazne, odnosno, ulazne grane, redom. Na kraju, lepimo čvorove sa identičnim oznakama.

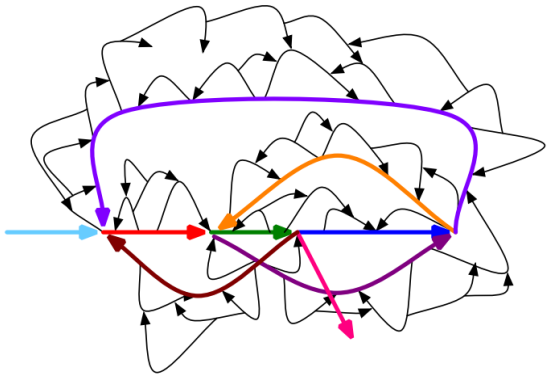
## 2.9 U realnosti

Ovde smo imali neke nerealne pretpostavke:

- Savršena pokrivenost genoma očitavanjima (svaki $k$-gram iz genoma je očitan). Očitavanja dužine 250 nukleotida dobijena *Illumina* tehnologijom predstavljaju samo mali deo 250-grama unutar genoma. Rešenje je u razbijanju dobijenih očitavanja na kraće $k$-grame.
- Očitavanja ne sadrže greške. U ovom slučaju, ako bismo razbili na manje $k$-grame, onda bismo dobili više niski koje imaju pogrešno očitavanje. Postavljamo pitanje kako se ovakvi slučajevi manifestuju u konstrukciji DeBrojnovog grafa. Dolazi do stvaranja *balončića* (engl. *bubble*) u grafu (videti sliku **??**). Jednostavan je slučaj kada govorimo o grešci na jednom očitavanju, međutim, ukoliko postoji više grešaka, onda dolazi do *eksplozije balončića* (videti sliku **??**).
- Rastojanja između očitavanja u okviru parova očitavanja su egzaktna.
- itd.




Slika 2.19: Primer pojave balončića u De Brojnovom grafu usled pojave greške u očitavanju nukleotida `T` nukleotidom `C`.



Slika 2.20: Eksplozija balončića.

## 2.10 Zadaci sa vežbi

U nastavku će biti predstavljeni zadaci sa vežbi na kursu rađeni u programskom jeziku Python.

### 2.10.1 Maximal Non Branching Path

```
1  from collections import deque
2  import copy
3
4  # Secenje DNK niske na k-mere
5  def string_to_k_mers(dna_string, k):
6      k_mers = []
7
8      for i in range(len(dna_string) - (k-1)):
9          k_mer = dna_string[i:i+k]
10         k_mers.append(k_mer)
11
12     return k_mers
13
14 # Konstruisanje Debruijn grafa od k-mera
15 def debruijn_graph_from_k_mers(k_mers):
16     G = {}
17
18     for k_mer in k_mers:
19         u = k_mer[:-1]
```

```
20          v = k_mer[1:]
21
22          if u in G:
23              if v not in G[u]:
24                  G[u].append(v)
25          else:
26              G[u] = [v]
27
28          if v not in G:
29              G[v] = []
30
31      return G
32
33
34  # Izracunavanje ulaznog i izlaznog stepena za zadati cvor
35  def degree(G, v):
36      out_deg = len(G[v])
37      in_deg = 0
38
39      for u in G:
40          if v in G[u]:
41              in_deg += 1
42
43      return (in_deg, out_deg)
44
45  # Pronalazenje izolovanih 1 in 1 out ciklusa u grafu polazeci od zadatog cvora
46  def isolated_cycle(G, v):
47      cycle = []
48
49      (in_deg, out_deg) = degree(G, v)
50
51      while in_deg == 1 and out_deg == 1:
52          u = G[v][0]
53          cycle.append((v,u))
54          if cycle[0][0] == cycle[-1][1]:
55              return cycle
56
57          v = u
58          (in_deg, out_deg) = degree(G, v)
59
60      return None
61
62
63  # Pronalazenje maksimalnih nerazgranatih putanja u grafu
64  def maximal_non_branching_paths(G):
65      paths = []
66      visited = {}
67
68      for v in G:
69
70          (v_in_deg, v_out_deg) = degree(G, v)
71          if v_in_deg != 1 or v_out_deg != 1:
72
```

```
73              visited[v] = True
74
75              if v_out_deg > 0:
76
77                  for w in G[v]:
78                      non_branching_path = [(v,w)]
79
80                      visited[w] = True
81                      (w_in_deg, w_out_deg) = degree(G, w)
82
83                      while w_in_deg == 1 and w_out_deg == 1:
84                          u = G[w][0]
85                          non_branching_path.append((w,u))
86                          w = u
87                          visited[w] = True
88                          (w_in_deg, w_out_deg) = degree(G, w)
89
90                      paths.append(non_branching_path)
91
92      for v in G:
93          if v not in visited:
94              c = isolated_cycle(G, v)
95              if c != None:
96                  paths.append(c)
97
98      return paths
99
100
101 # Konstruisanje DNK niske od dobijene putanje
102 def create_string_from_path(path):
103
104     dna_string = path[0][0]
105
106     for i in range(len(path)):
107         dna_string += path[i][1][-1]
108
109     return dna_string
110
111
112 def main():
113     dna_string = "AATCGTGACCTCAACT"
114     #               TCGTGAC
115     #             AATC
116     #                         ACT
117     #                   ACCT
118     #                       AAC
119     #                    TCAAC
120     k = 3
121     k_mers = string_to_k_mers(dna_string, k)
122     g = debruijn_graph_from_k_mers(k_mers)
123     paths = maximal_non_branching_paths(g)
124
125     print(paths)
```

```

if __name__ == "__main__":
    main()
```

### 2.10.2 All Euler Cycles

```
from collections import deque
import copy

# Izracunavanje ulaznog i izlaznog stepena za zadati cvor
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)

# Pronalazenje izolovanih 1 in 1 out ciklusa u grafu polazeci od zadatog cvora
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

# Konstruisanje DNK niske od dobijene putanje
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


# Pronalazenje cvorova od kojih postoje grane ka zadatom cvoru v
def incoming(G, v):
    in_list = []

    for u in G:
```

```
48          if v in G[u]:
49              in_list.append(u)
50
51      return in_list
52
53  # Pronalazenje cvorova do kojih postoje grane od zadatog cvora v
54  def outgoing(G, v):
55      return G[v]
56
57
58  # Pravljenje (u,v,w) "zaobilaznice" u zadatom grafu G
59  def bypass(G, u, v, w):
60      G_p = copy.deepcopy(G)
61      G_p[u].remove(v)
62      G_p[v].remove(w)
63      G_p[u].append(v+"'")  #v'
64      G_p[v+"'"] = [w]
65      return G_p
66
67
68  def DFS(G, v, visited):
69      visited[v] = True
70
71      for w in G[v]:
72          if w not in visited:
73              DFS(G, w, visited)
74
75
76  # Provera da li je graf povezan u odnosu na DFS obilazak iz zadatog cvora
77  def is_connected(G):
78
79      visited = {};
80      for v in G:
81          DFS(G,v,visited)
82          break;
83
84      for v in G:
85          if v not in visited:
86              return False
87
88      return True
89
90  # Pronalazenje svih Ojlerovih ciklusa u zadatom grafu G
91  def all_eulerian_cycles(G):
92      all_graphs = deque([copy.deepcopy(G)])
93      cycles = []
94
95      while len(all_graphs) > 0:
96          G_p = all_graphs.popleft()
97          v_p = None
98          for v in G_p:
99              (in_deg, out_deg) = degree(G_p, v)
100
```

```
101             if in_deg > 1:
102                 v_p = v
103                 break
104 
105         if v_p != None:
106             for u in incoming(G_p, v_p):
107                 for w in outgoing(G_p, v_p):
108                     new_graph = bypass(G_p, u, v, w)
109                     if is_connected(new_graph):
110                         all_graphs.append(copy.deepcopy(new_graph))
111         else:
112             for k in G_p:
113                 cycle = isolated_cycle(G_p, k)
114                 if cycle != None:
115                     path = create_string_from_path(cycle)
116                     if path not in cycles:
117                         cycles.append(path);
118 
119     return cycles
120 
121 
122 
123 
124 def main():
125     G = {'AT' : ['TC'], 'TC' : ['CG'], 'CG': ['GA','GG'], 'GA':['AT','AC'], 'AC
    ↪ ':['CG'], 'GG':['GA']}
126 
127     print(all_eulerian_cycles(G))
128 
129 if __name__ == "__main__":
130     main()
```

### 2.10.3 String Spelled By Gapped Patterns

```
1  # Sastavljanje DNK niske pomocu k-mera
2  def string_spelled_by_patterns(patterns, k):
3      dna_string = patterns[0][:-1]
4  
5      for i in range(0, len(patterns)):
6          dna_string += patterns[i][-1]
7  
8      return dna_string
9  
10 # Sastavljanje DNK niske pomocu parova k-mera na udaljenosti d
11 def string_spelled_by_gapped_patterns(gapped_patterns, k, d):
12     first_patterns = [s[0] for s in gapped_patterns]
13     second_patterns = [s[1] for s in gapped_patterns]
14 
15     prefix_string = string_spelled_by_patterns(first_patterns, k)
16     suffix_string = string_spelled_by_patterns(second_patterns, k)
17 
18     print(prefix_string)
19     print(suffix_string)
```

```
20
21      for i in range(k+d, len(prefix_string)):
22          if prefix_string[i] != suffix_string[i-k-d]:
23              print('There is no string spelled by the gapped patterns')
24              return ''
25      return prefix_string + suffix_string[-k-d:]
26
27
28  def main():
29      gapped_patterns = [('CTG','CTG'),('TGA','TGA'),('GAC','GAC'),('ACT','ACT')]
30
31      print(string_spelled_by_gapped_patterns(gapped_patterns, 3, 1))
32
33  if __name__ == "__main__":
34      main()
```

# Glava 3

# Kako sekvenciramo antibiotike?

U ovom poglavlju i dalje govorimo o sekvenciranju, ali ćemo proširiti pogled i pokazati različite načine za sekvenciranje peptida.

## 3.1 Otkriće antibiotika

Pre svega, krenućemo sa biološkim uvodom. Šta su to antibiotici? Sama reč antibiotik znači ,,onaj koji ubija život", a tačnije, on predstavlja supstancu koja ubija bakterije. Kada ostavimo pomorandžu dugo negde gde je toplo, ona će da razvije čudne osobine kao što je buđ. Šta to znači? Buđ jeste jedna vrsta antibiotika što znači da se antibiotici nalaze u prirodi i da ih proizvode organizmi iz porodice gljiva (npr. buđi) i bakterija.

Mi ćemo posmatrati antibiotike na molekularnom nivou koji nam govori od čega su oni zapravo izgrađeni. Od svih antibiotika posmatraćemo ***tirocidin B1***, antibiotik koji proizvodi bakterija *Bacillus Brevis*. Tirocidin B1 na molekulskom nivou pripada *peptidima*, kratkim niskama aminokiselina, odnosno malim proteinima. Ovo je skok u odnosu na ono što smo do sada posmatrali – nukleotidne niske nad četvorostrukom azbukom $\Sigma = \{A, C, G, T\}$, odnosno DNK. Za DNK smo govorili da se pojavljuje u svakoj ćeliji svakog živog bića i da je veoma značajna supstanca jer sadrži recept (tačnije, nosi informaciju) za pravilno funkcionisanje i razvoj svakog živog bića. Da bi se svako živo biće pravilno razvijalo, neophodno je da njegove ćelije proizvode (sintetišu) u tačno određeno vreme određene supstance koje se nazivaju *proteini*. DNK nosi informaciju o tome kako treba neki protein da izgleda, od čega treba da se sastoji. Zašto je to bitno? Na primer, kada je dan, neke biljke treba da vrše fotosintezu, a za vršenje fotosinteze treba u samim ćelijama biljaka da se sintetišu određeni proteini.

Proteini su, nakon nukleinskih kiselina, druga značajna grupa molekula koja sa računarske tačke gledišta takođe predstavlja dugačke niske, ali ne nad azbukom od 4 karaktera, nego nad azbukom od 20 karaktera, a svaki karakter predstavlja molekul koji se naziva *aminokiselina*. Kao i nukleinske kiseline, aminokiseline se predstavljaju velikim latiničnim slovima $\{V, K, L, F, P, W, N, Q, Y, G, A, I, M, D, E, S, T, C, R, H\}$, a pored toga postoje i troslovne oznake $\{Val, Lys, Leu, Phe, Pro, Trp, Asn, Gln, Tyr, Gly, Ala, Ile, Met, Asp, Glu, Ser, Thr, Cys, Arg, His\}$. U prirodi postoji mnogo više od 20 aminokiselina, ali 20 njih najčešće učestvuje u sastav proteina. DNK upravlja time kada će nastati protein u okviru ćelije. Recept za nastajanje svakog proteina je zapisan u DNK. Kako je taj recept zapisan, videćemo u nastavku.

Proteini se još nazivaju i *polipeptidi*. Dužina proteina je obično od 100 aminokiselina do nekoliko hiljada (proteini su kraći od genomske sekvence). Tirocidin B1 je peptid jer se sastoji iz malog broja aminokiselina, svega deset – $V, K, L, F, P, W, F, N, Q, Y$. Problem sekvenciranja antibiotika jeste problem određivanja aminokiselina koje ulaze u sastav tog antibiotika. U prethodnom poglavlju smo sekvencirali genom, ali tehnike iz prethodnog poglavlja nećemo moći da koristimo u sekvenciranju tirocidina B1, što će biti objašnjeno u poglavlju **??**.

## 3.2 Kako bakterije prave antibiotike?

Pre rešavanja problema sekvenciranja antibiotika, govorićemo o zanimljivoj i kompleksnoj temi, a to je tema – kako se prave proteini? Već je pomenuto da se u okviru DNK nalazi recept za pravljenje proteina. Sada je vreme da se zapitamo kako je sve to zapisano u DNK pomoću $A, C, G, T$.

Znamo da je DNK dvostruki lanac čiji su krajevi označeni sa 5' i 3' (uvek čitamo lanac od 5' ka 3'). DNK jeste jedna vrsta nukleinskih kiselina koje postoje u ćeliji živih bića. Pored nje, postoje i različite vrste **ribonukleinskih kiselinina**, odnosno **RNK**. Ribonukleinske kiseline nisu predstavljene dvostrukim lancem, već jednostrukim. One se sastoje od nukleotida $A, C, G, U$. Umesto timina, kod RNK se pojavljuje nukleotid uracil koji se označava sa $U$.

DNK se ***prepisuje*** u RNK. Šta to znači? Da bi nastali proteini, neophodno je da se na osnovu dva lanca od DNK konstruiše RNK molekul. Pošto se RNK molekul sastoji od istih nukleotida kao i DNK, osim timina, onda kažemo da formiranje RNK na osnovu DNK predstavlja jednostavno *prepisivanje* nukleotida iz oba lanca DNK, uz zamenu nukleotida $T$ sa nukleotidom $U$. Drugi naziv za prepisivanje jeste *transkripcija*. Ovo je prvi korak, i dalje nismo došli do aminokiseline, i dalje smo u azbuci nukleotida. RNK predstavlja jedan međukorak između DNK i samog proteina.

Drugi korak jeste *prevođenje*, odnosno *translacija*, prepisanog RNK u proteine. Imamo 4 nukleotida $A, C, G, U$ i treba njih da prevedemo u nisku od 20 mogućih aminokiselina. To znači da mora da postoji neko preslikavanje, nekakav kod koji prevodi neke $k$-grame nukleotida u aminokiseline. Nad azbukom od 4 nukleotida postoji 16 različitih 2-grama, tj. bigrama. Da li možemo tih 16 bigrama da preslikamo u 20 aminokiselina? Tačnije, da li dva nukleotida možemo da preslikamo u jednu aminokiselinu? Ne možemo, jer moramo za svaku aminokiselinu da znamo koji je bigram označava. Pošto ne možemo to da uradimo sa bigramima, da li možemo sa 3-gramima? Svih mogućih 3-grama nad azbukom od 4 nukleotida ima 64. To znači da će svaka od aminokiselina imati svoj kod, a neke od njih će možda imati i više kodova, tj. više trigrama može da ukazuje na jednu aminokiselinu. To je u redu, bitno je da je naša funkcija „*na*", ne mora da bude „$1 - 1$". Ali kako napraviti funkciju? Ne možemo svojevoljno da dodelimo trigramima određene aminokiseline. Ta funkcija je unapred utvrđena, odnosno, prirodom determinisana i dokazana. U nastavku, koristićemo drugačiji naziv za naše 3-grame.

**Definicija 3.1.** ***Kodon*** *predstavlja jedan 3-gram (triplet) nukleotida.*

Preslikavanje o kojem je do sada bilo reči se naziva *genetski kod* i on je prikazan na slici **??**.

**Definicija 3.2.** ***Genetski kod*** *predstavlja preslikavanje skupa kodona u skup aminokiselina.*

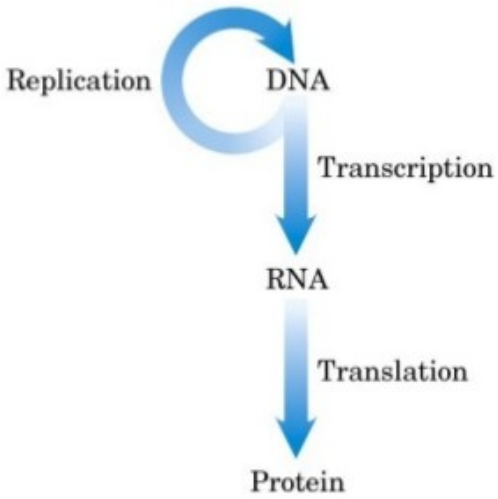
Vidimo da se, na primer, kodon $UGG$ preslikava u aminokiselinu $Trp(W)$, dok se više kodona, $CUA, CUC, CUG, CUU, UUA, UUG$, preslikava u jednu aminokiselinu $Lys(L)$.

Redom, kodone iz RNK preslikavamo u aminokiseline. Ali, kako znamo da je kraj nekog proteina? U genetskom kodu je i tako nešto kodirano. Postoje tzv. **stop kodoni** koji označavaju da iza njih nema više aminokiselina koje čine taj protein. Ti stop kodoni su $UAA, UAG, UGA$.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | AAA | K | 16 | CAA | Q | 32 | GAA | E | 48 | UAA | * |
| 1 | AAC | N | 17 | CAC | H | 33 | GAC | D | 49 | UAC | Y |
| 2 | AAG | K | 18 | CAG | Q | 34 | GAG | E | 50 | UAG | * |
| 3 | AAU | N | 19 | CAU | H | 35 | GAU | D | 51 | UAU | Y |
| 4 | ACA | T | 20 | CCA | P | 36 | GCA | A | 52 | UCA | S |
| 5 | ACC | T | 21 | CCC | P | 37 | GCC | A | 53 | UCC | S |
| 6 | ACG | T | 22 | CCG | P | 38 | GCG | A | 54 | UCG | S |
| 7 | ACU | T | 23 | CCU | P | 39 | GCU | A | 55 | UCU | S |
| 8 | AGA | R | 24 | CGA | R | 40 | GGA | G | 56 | UGA | * |
| 9 | AGC | S | 25 | CGC | R | 41 | GGC | G | 57 | UGC | C |
| 10 | AGG | R | 26 | CGG | R | 42 | GGG | G | 58 | UGG | W |
| 11 | AGU | S | 27 | CGU | R | 43 | GGU | G | 59 | UGU | C |
| 12 | AUA | I | 28 | CUA | L | 44 | GUA | V | 60 | UUA | L |
| 13 | AUC | I | 29 | CUC | L | 45 | GUC | V | 61 | UUC | F |
| 14 | AUG | M | 30 | CUG | L | 46 | GUG | V | 62 | UUG | L |
| 15 | AUU | I | 31 | CUU | L | 47 | GUU | V | 63 | UUU | F |

Slika 3.1: Genetski kod.

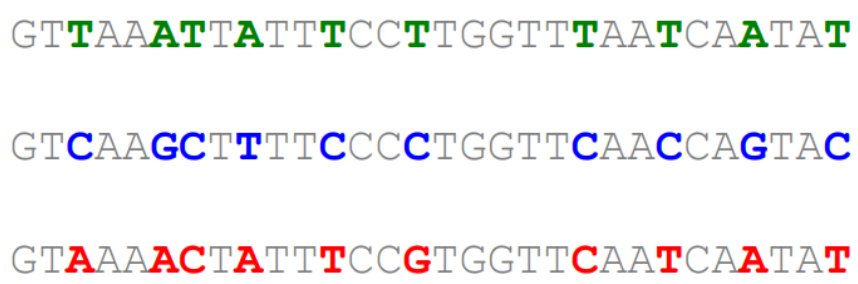
Dolazimo do jednog veoma značajnog biološkog aksioma – **centralne dogme molekularne biologije**. Ona govori da se transkripcijom na osnovu DNK može dobiti RNK, a translacijom se iz RNK, na osnovu genetskog koda, dobijaju proteini. Ovu teoriju je predstavio Fransis Krik (eng. *Francis Crick*) i prikazana je na slici **??**.




Slika 3.2: Centralna dogma molekularne biologije.

Ono što želimo da saznamo jeste koje aminokiseline i kojim redom ulaze u sastav našeg malog peptida tirocidina B1. Pošto se on sastoji iz 10 aminokiselina, to znači da ga čine 30 nukleotida u genomu bakterije *Bacillus Brevis* koje će da se prepišu u RNK i da se prevedu iz RNK u tirocidin B1. Hiljade različitih 30-grama se može prevesti u tirocidin B1 jer se u genetskom kodu različiti kodoni mogu prevesti u istu aminokiselinu. Na slici **??** su prikazani neki od takvih 30-grama. Vidimo da oni nisu previše slični.

Treba uzeti u obzir da translacija može početi na bilo kojoj poziciji u genomu. To znači




Slika 3.3: Neki od 30-grama koji se mogu prevesti u tirocidin B1.

da bismo za datu poziciju, ako gledamo 30-gram, mogli da imamo 6 različitih tzv. **čitajućih okvira**, tj. 6 varijanti prepisivanja u RNK i onda prevođenja. Tri čitajuća okvira potiču iz tri nukleotida iz jednog kodona iz jednog prevedenog RNK lanca (ako krenemo da čitamo od prvog nukleotida, to je jedan čitajući okvir, iz drugog nukleotida je drugi čitajući okvir, iz trećeg nukleotida je treći čitajući okvir, a ako pročitamo od četvrtog nukleotida, to je već isti čitajući okvir kao prvi jer tu kreće novi kodon), a isto tako za drugi RNK lanac imamo tri čitajuća okvira sa druge strane.

Naš peptid tirocidin B1 jeste *cikličan*. Tih 10 aminokiselina koje ga čine idu nekim redom, ali su one povezane u krug, tako da imamo ukupno 10 različitih **linearnih reprezentacija** za tirocidin B1 u zavisnosti od toga koja nam je prva aminokiselina bila u samom receptu DNK. Koju god linearnu reprezentaciju pronađemo, rešili smo problem.

Ne odustajemo od pronalaženja 30-grama u genomu bakterije *Bacillus Brevis* koji kodira bar jednu linearnu reprezentaciju od svih 10 koje čine tirocidin B1. Pretpostavimo da imamo na raspolaganju veoma moćan računar i neograničeno vreme. Doći ćemo do jednog čudnog rezultata. Nećemo uspeti da pronađemo nijedan 30-gram u genomu bakterije *Bacillus Brevis* koji kodira bar jednu linearnu reprezentaciju proteina tirocidina B1. Zašto? Stvari se komplikuju. Na ovom primeru je pokazano da centralna dogma ne važi uvek, odnosno ne važi da svaki protein u ćeliji nastaje na osnovu recepta koji je zapisan u DNK. Centralna dogma govori da se proces transkripcije izvršava pod uticajem enzima koji se zove *RNK polimeraza*, a translacija RNK u protein se vrši u ćelijskoj organeli koja se naziva *ribozom*. Postoje neki proteini koji ne nastaju na ovaj način, nego na specijalan način gde obično sekvenciranje genoma ne može da nam pomogne. Moramo da predložimo novi metod kako možemo da pronađemo odgovarajuću sekvencu aminokiselina.

Edvard Tejtum (eng. *Edward Tatum*), jedan od poznatih američkih genetičara, je 1963. godine inhibirao ribozom bakterije *Bacillus Brevis*. Šta ovo znači? S obzirom da se znalo da se u ribozomu vrši translacija RNK u protein, on je onemogućio da se bilo šta desi u ribozomu, isključio je funkcionisanje te organele u ćeliji i očekivao je da se neće stvoriti nijedan protein, pa ni tirocidin B1. Međutim, suprotno očekivanjima, nastavljena je proizvodnja nekih peptida, uključujući i tirocidine. Ovo je bilo izuzetno iznenađujuće otkriće.

Fric Lipman (eng. *Fritz Lipmann*), američko-nemački biohemičar, je 1969. godine pokazao da tirocidini spadaju u grupu **ne-ribozomalnih peptida (NRP-ova)**. To su peptidi za čiju sintezu nisu odgovorni ribozomi i RNK polimeraza već enzimi poznati pod nazivom **NRP sintetaze**, molekuli koji se takođe nalaze u ćeliji i utiču na različite procese koji se dešavaju u njoj. To znači da se stvaranje tirocidina razlikuje od većeg broja proteina u živim bićima.

Kako izgleda sinteza tirocidina B1 pomoću NRP sintetaze? Postoji veliki broj različitih NRP sintetaza, nije samo jedna odgovorna za stvaranje svih mogućih NRP-ova, nego za svaki ne-ribozomalni peptid postoji odgovarajuća NRP sintetaza. Ona NRP sintetaza koja je odgo-

vorna za stvaranje tirocidina B1 se sastoji od 10 različitih podjedinica koje nazivamo *moduli*. Svaki modul je odgovoran za nadovezivanje jedne aminokiseline na budući molekul tirocidina B1. Svaki od modula privuče jednu aminokiselinu i spoji je sa prethodnom, a poslednji korak jeste cirkularizacija – spajanje aminokiselina nastalih uz pomoć prvog i poslednjeg modula radi kreiranja cikličnog peptida.

## 3.3 Sekvenciranje antibiotika razbijanjem na komade

Pošto nam sekvenciranje genomske sekvence i pronalaženje odgovarajuće podniske koja je zadužena za translaciju u aminokiseline odgovarajućeg peptida ne može pomoći u sekvenciranju tirocidina B1 (jer on ne nastaje na osnovu informacije zapisane u DNK), postavljamo pitanje da li postoji način na koji možemo da sekvenciramo antibiotike. Moramo direktno da sekvenciramo peptid. Jedan od načina jeste **sekvenciranje razbijanjem na komade** i biće predstavljen u ovoj sekciji.

U sekvenciranju antibiotika može nam pomoći mašina koja se naziva **maseni spektrometar** i koju možemo opisati kao skupu molekularnu vagu. Šta on radi? Za početak ćemo da se zapitamo kako možemo da merimo težinu, tačnije masu molekula.

Pošto se molekuli sastoje od atoma, prvo treba da govorimo o masi pojedinačnog atoma. U atomima postoje protoni, neutroni i elektroni. Protoni i neutroni su približno iste mase, dok su elektroni izuzetno mali i gotovo zanemarljive mase. Zato masu jednog atoma možemo svesti na masu protona, odnosno neutrona koji učestvuju u izgradnji konkretnog atoma koji posmatramo, pa se i masa molekula može izračunati kao suma masa atoma koji ga čine. Maseni spektrometar vraća masu molekula izračunatu u **Daltonima**.

$$1 \quad Dalton(Da) \approx masa \quad jednog \quad protona/neutrona$$
$$Masa \quad molekula \approx suma \quad masa \quad protona/neutrona$$

Posmatrajmo masu jedne aminokiseline, recimo glicina. Glicin ima hemijsku formulu $C_2H_3ON$. Ugljenik ima masu 12, vodonik ima masu 1, kiseonik 16, a azot 14. Na osnovu ovih masa računamo masu celog molekula glicina.

$$masa(C_2H_3ON) = 12 * 2 + 1 * 3 + 16 * 1 + 14 * 1 \approx 57Da$$

Simbol $\approx$ koristimo jer nam je stvarna masa nešto malo drugačija od celobrojne mase koju dobijamo ovde. Stvarna masa glicina iznosi $57.02Da$. Podrazumevaćemo da je masa celog molekula upravo celobrojna masa koju smo dobili.

Tabela masa svih aminokiselina data je u tabeli **??**.

Tabela 3.1: Tabela masa 20 aminokiselina poređanih rastuće prema celobrojnim masama.

| G | A | S | P | V | T | C | I,L | N | D | K,Q | E | M | H | F | R | Y | W |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 57 | 71 | 87 | 97 | 99 | 101 | 103 | 113 | 114 | 115 | 128 | 129 | 131 | 137 | 147 | 156 | 163 | 186 |

Primećujemo da neke aminokiseline imaju iste mase, npr. $I$ i $L$, $K$ i $Q$, pa za 20 aminokiselina imamo 18 celobrojnih masa.

Kada imamo mase aminokiselina, možemo da se zapitamo koja je masa tirocidina B1. Znajući da se tirocidin B1 sastoji iz 10 aminokiselina ovim redom: $VKLFPWFNQY$, masu računamo koristeći tabelu **??**.

$$masa(tirocidina \quad B1) = 99 + 128 + 113 + 147 + 97 + 186 + 147 + 114 + 128 + 163 = 1322$$

Vratimo se na maseni spektrometar o kojem smo govorili ranije. Zamislimo da imamo kratak peptid koji se sastoji od samo 4 aminokiseline $NQEL$. Uzorak ovog peptida ubacimo u maseni spektrometar. Šta se u njemu dešava? U njemu se nekim hemijskim procesima, u čije detalje nećemo ulaziti, generišu svi podpeptidi ulaznog peptida. Sa računarske tačke gledišta, maseni spektrometar generiše od zadate niske $NQEL$ sve moguće podniske, podrazumevajući da je ulazni peptid cikličan. Tako se generišu podniske dužine jedan: $N, Q, E, L$, podniske dužine dva: $NQ, QE, EL, LN$ i podniske dužine tri: $NQE$, $QEL$, $ELN$, $LNQ$. Maseni spektrometar za svaki od dobijenih podpeptida može da odredi molekulsku masu. Ono što mi dobijemo kao izlaz iz masenog spektrometra nisu podpeptidi, mi ne znamo koje podpeptide je dobio, niti kojim podpeptidima je prudružena koja masa. Izlaz iz masenog spektrometra jeste samo niz masa! Taj izlazni niz može da sadrži i dva ista broja jer neki podpeptidi mogu da imaju istu masu.

Kako bismo formulisali računarski problem, moramo da definišemo šta je to *teorijski spektar*.

**Definicija 3.3.** ***Teorijski spektar peptida*** *predstavlja niz masa svih mogućih podpeptida tog peptida, uključujući nulu kao masu praznog peptida i masu samog peptida.*

Poznajući sastav peptida, lako možemo da izračunamo teorijski spektar. Suprotan smer je težak, odnosno teško je da na osnovu teorijskog spektra zaključimo kako je izgledao peptid. Upravo ovo jeste problem sekvenciranja ciklopeptida.

**Problem 7** (Problem sekvenciranja ciklopeptida)**.** *Rekonstruisati ciklični peptid na osnovu njegovog teorijskog spektra.*

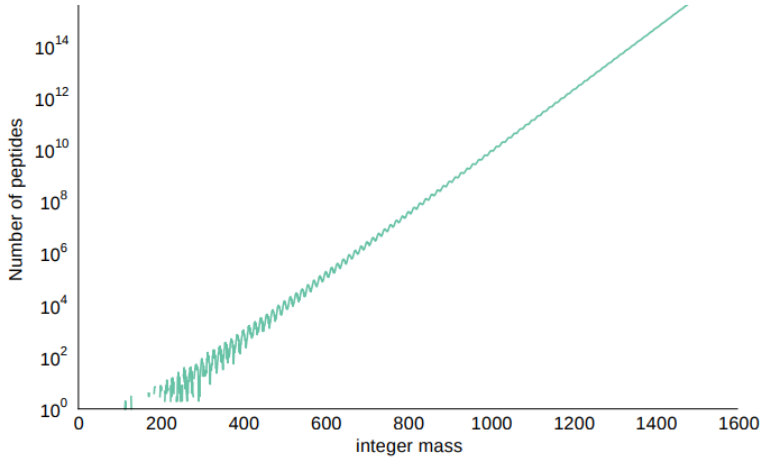
### 3.3.1 Sekvenciranje ciklopeptida grubom silom

Kada dobijemo spektar iz masenog spektrometra, najveća masa će označavati masu celog peptida. Želimo prvo da generišemo sve peptide sa masom jednakom masi celog peptida, zatim da za svaki tako generisan peptid formiramo teorijski spektar i uporedimo ga sa datim spektrom. Algoritam grube sile za problem sekvenciranje ciklopeptida dat je u nastavku.

```
1 BFCyclopeptideSequencing(Spectrum)
2 begin
3     // ParentMass(Spectrum) jeste najveca masa u spektru Spectrum
4     for every Peptide with Mass(Peptide) equal to ParentMass(Spectrum)
5         if Spectrum == CycloSpectrum(Peptide)
6             output Peptide
7 end
```

Vidimo da u ovom algoritmu ispitujemo sve kandidate (peptide sa istom masom kao dati peptid). Koliko imamo takvih kandidata? Grafik koji oslikava odgovor na ovo pitanje dat je na slici **??**.

Vidimo da je ovaj algoritam grube sile eksponencijalne složenosti. Pod uslovom da imamo dovoljno brz računar, ovako nešto bismo i mogli da izračunamo. Ali, šta bi bili nedostaci ovog algoritma grube sile? Možemo da imamo dva peptida sa istom masom, a da su potpuno različiti. Na primer, peptid $NQEL$ i peptid $TMDH$ imaju masu 484. Kako možemo da isključimo pogrešan peptid? Za oba ova peptida možemo da generišemo teorijski spektar. Ispostavlja se da su njihovi spektri potpuno različiti. Želimo da ovu informaciju iskoristimo u sledećem pristupu rešavanja problema sekvenciranja ciklopeptida. Cilj nam je da ne idemo grubom silom već da ogroman broj kandidata od samog početka odstranimo.

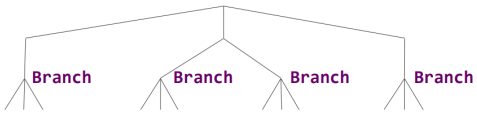



Slika 3.4: Broj peptida sa zadatom masom.

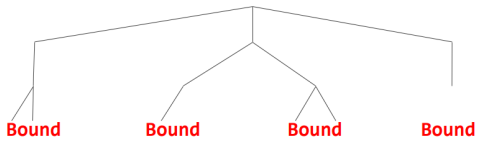
### 3.3.2 *Branch-and-Bound* algoritam za sekvenciranje ciklopeptida

U ovom pristupu postepeno konstruišemo kandidate za rešenja od manjih linearnih peptida za razliku od prethodnog pristupa u kome smo odmah generisali ceo peptid koji postaje kandidat. Na taj način ćemo smanjiti ukupan broj linearnih peptida koje posmatramo. Ovakav pristup se koristi kod ***Branch-and-Bound* algoritama** koji će biti opisani u nastavku.

Kod *Branch-and-Bound* algoritama u celokupnom prostoru svih mogućih rešenja vršimo nekakva odsecanja. Počnemo od kratkog peptida dužine jedan, pa ga proširimo na sve moguće načine, odnosno, dodamo po jednu aminokiselinu i od toga napravimo sve moguće kandidate dužine dva. To je *branch grana* i predstavljena je na slici **??**. *Bound grana* bi od postojećih kandidata, nastalih u *branch* granama, isključila neke potencijalne kandidate. *Bound* grana je predstavljena na slici **??**. Postupak proširivanja i odsecanja ponavljamo sve dok ne dođemo do odgovarajućih vrednosti. Na ovaj način će nam ostati mnogo manje kandidata za rešenja nego u prethodnom pristupu grube sile.




Slika 3.5: *Branch* grane algoritma *Branch-and-Bound*.




Slika 3.6: *Bound* grane algoritma *Branch-and-Bound*.

Primenimo ovaj algoritam na konkretan problem. Recimo da nam je dat spektar

0 97 97 99 101 103 196 198 198 200 202 295 297 299 299 301 394 396 398 400 400 497.

Vidimo da se u datom spektru nalaze mase nekih aminokiselina, što nam govori koje aminokiseline ulaze u sastav traženog peptida. Te aminokiseline su $P, V, T, C$ sa masama $97, 99, 101, 103$. Ovo znači da možemo da počnemo ne sa svih 20 aminokiselina, nego sa 4 unigrama $P, V, T, C$. Ovo je unapred jedna *bound* grana jer smo 20 aminokiselina sveli na 4 kandidata aminokiselina. Zatim idemo na branch granu, širimo unigrame u sve moguće bigrame: $PA$, $PC$, $PD$,..$PY$, $VA$, $VC$, $VD$,...,$VY$, $TA$, $TC$, $TD$,..., $TY$, $CA$, $CC$, $CD$,..., $CY$. Proširujemo sa svih 20 aminokiselina jer ćemo kasniji videti da ovako zadati spektar jeste čisto teorijski spektar, a maseni spektrometar skoro nikada u praksi ne vraća teorijski spektar već spektar sa nekakvim greškama. Kako možemo

da skratimo ovu listu, kako možemo da izvršimo korak *bound* u ovom trenutku? Posmatramo da li postoje odgovarajući bigrami koji se takođe pojavljuju u spektru. Zbog toga uvodimo pojam **konzistentnosti**.

**Definicija 3.4.** *Za proizvoljan podpeptid $p_1, .., p_n$ kažemo da je **konzistentan** sa spektrom $S$ ukoliko se svaka masa iz teorijskog spektra podpeptida $p_1, .., p_n$ nalazi u spektru $S$.*

Na primer, $PV$ je **konzistentno** sa spektrom ukoliko se masa od $P$, masa od $V$ i masa od $PV$ nalaze u spektru.

Konzistentnost ćemo koristiti u *bound* koraku, tačnije, izbacićemo sve bigrame koji nisu konzistentni sa spektrom. Tako dobijemo listu konzistentnih bigrama $PV$, $PT$, $PC$, $VP$, $VT$, $VC$, $TP$, $TV$, $CP$, $CV$ koju proširujemo u sve moguće 3-grame, a zatim svodimo na listu samo konzistentnih 3-grama. Postupak ponavljamo. Kada dođemo do liste konzistentnih pentagrama $PVCPT$, $PTPVC$, $PTPVC$, $PCVPT$, $VPTPC$, $VCPTP$, $TPVCP$, $TPCVP$, $CPTPV$, $CVPTP$ vidimo da zapravo svi oni pokazuju na jedan isti ciklični peptid.

Pseudokod opisanog algoritma dat je u nastavku.

```
1  CyclopeptideSequencing(Spectrum)
2  begin
3      Peptides = a set containing only the empty peptide
4      while Peptides is non-empty
5          // prosirujemo sve peptide u skupu sa svim mogucim aminokiselinama
6          Peptides = Expand(Peptides)
7          for each Peptide in Peptides
8              // ParentMass(Spectrum) jeste najveca masa u spektru
9              if Mass(Peptide) = ParentMass(Spectrum)
10                 if Cyclospectrum(Peptide) = Spectrum
11                     output Peptide
12                 remove Peptide from Peptides
13             else if Peptide is not consistent with Spectrum
14                 remove Peptide from Peptides
15 end
```

Podsetimo se da je složenost algoritma grube sile, koji ovde pokušavamo da poboljšamo, eksponencijalna. Ispostavlja se da *Branch-and-Bound* algoritam takođe može biti eksponencijalne složenosti za neke peptide, ali je u praksi veoma brz.

## 3.4 Prilagođavanje sekvenciranja za spektre sa greškama

Spektar koji smo do sada definisali jeste teorijski spektar. Za razliku od njega, spektar koji izlazi iz masenog spektrometra, **eksperimentalni spektar**, često sadrže greške. O kakvim greškama se govori biće prikazano pomoću slike **??**.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **teorijski:** | 0 | | 113 | 114 | 128 | 129 | 227 | 242 | 242 | 257 | | 355 | 356 | 370 | 371 | 484 |
| **eksperimentalni:** | 0 | 99 | 113 | 114 | 128 | | 227 | | | 257 | 299 | 355 | 356 | 370 | 371 | 484 |

Slika 3.7: Primer teorijskog i eksperimentalnog spektra za $NQEL$.

**Lažne mase** jesu mase koje su na slici **??** prikazane zelenom bojom. To su mase koje su prisutne u eksperimentalnom spektru, ali nisu prisutne u teorijskom spektru.

**Nedostajuće mase** jesu mase koje su na slici **??** prikazane plavom bojom. To su mase koje se nalaze u okviru teorijskog spektra, ali ne i u okviru eksperimentalnog spektra.

Zbog pojave ovih otežavajućih okolnosti, tj. grešaka u spektru, neophodan je novi algoritam jer se kod dva predložena algoritma teorijski spektar peptida morao u potpunosti poklapati sa spektrom peptida koji je predstavljao rešenje problema. Sada moramo da olabavimo taj uslov pa uvodimo pojam **skor peptida**.

**Definicija 3.5.** ***Skor peptida*** *pokazuje koliko masa njegov teorijski spektar deli sa eksperimentalnim spektrom.*

Tako, za sliku **??**, skor iznosi 11. Želimo da skor bude što veći.

S obzirom da imamo nov način upoređivanja, moramo da unapredimo naš *Branch-and-Bound* algoritam, konkretno korak odsecanja.

Uzmimo primer golfa. U golfu, kada igrači prođu prvi krug takmičenja, u sledeći krug prolaze dalje samo igrači koji su konkurentni, oni koji imaju šanse da nešto osvoje. To znači da možemo da kažemo da nam je odsecanje takvo da, na primer, prva tri igrača sa najboljim skorom idu dalje, a ukoliko imamo još neke igrače koji imaju isti skor kao poslednji igrač, onda i oni prolaze dalje. Znači, zadržavaju se tri najbolja igrača *„with ties"*. Ovakav sistem primenjen na *Branch-and-Bound* algoritam prikazan je u sledećem pseudokodu.

```
1  LeaderboardCyclopeptideSequencing(Spectrum, N)
2  begin
3      Leaderboard = set containing only the empty peptide
4      LeaderPeptide = empty peptide
5
6      while Leaderboard is non-empty
7          // prosirujemo sve elemente koji se nalaze u okviru skupa Leaderboard
8          Leaderboard = Expand(Leaderboard)
9          for each Peptide in Leaderboard
10             // ParentMass(Spectrum) predstavlja najvecu masu u spektru Spectrum
11             if Mass(Peptide) == ParentMass(Spectrum)
12                 if Score(Peptide, Spectrum) > Score(LeaderPeptide, Spectrum)
13                     LeaderPeptide = Peptide
14             else if Mass(Peptide) > ParentMass(Spectrum)
15                 remove Peptide from Leaderboard
16         // odsecamo kandidate iz Leaderboard na osnovu njihovog skora
17         Leaderboard = Trim(Leaderboard, Spectrum, N)
18
19     output LeaderPeptide
20 end
21
22 Trim(Leaderboard, Spectrum, N, AminoAcid, AminoAcidMass)
23 begin
24     for j=1 to |Leaderboard|
25         Peptide = j-th peptide in Leaderboard
26         // LinearScore jeste skor nad linearnim spektrom
27         LinearScores[j] = LinearScore(Peptide, Spectrum)
28
29     sort Leaderboard according to the dec order of scores in LinearScores
30     sort LinearScores in dec order
31
32     for j=N+1 to |Leaderboard|
33         if LinearScores[j] < LinearScores[N]
34             remove all peptides starting from the j-th peptide from Leaderboard
35             return Leaderboard
36
```

```
37      return Leaderboard
38  end
```

*Leaderboard* pristup omogućava da bolje definišemo za esperimentalni spektar kod *Branch-and-Bound* algoritma onu bound fazu kada treba da izbacimo neke kandidate.

### 3.4.1 Testiranje na spektru tirocidina B1

U ovom delu razmatraćemo rezultate testiranja na $Spectrum_{10}$, spektru sa 10% lažnih/nedostajućih masa.

Kada primenimo *LeaderboardCyclopeptideSequencing* na spektar sa 10% loših vrednosti, tada zaista dobijemo peptid sa najvišim skorom $VKLFPWFNQY$ koji odgovara tirocidinu B1. Međutim, ukoliko uzmemo spektar $Spectrum_{25}$ koji ima 25% lažnih i nedostajućih vrednosti, spektar koji se još više udaljava od teorijskog spektra, onda se peptid sa najvišim skorom $VKLFPADFNQY$ razlikuje od peptida $VKLFPWFNQY$ koji želimo da dobijemo.

Ovo znači da *LeaderboardCyclopeptideSequencing* algoritam radi dobro kada nam je eksperimentalni spektar malo različit od teorijskog.

## 3.5 Od 20 do više od 100 aminokiselina

U ovoj sekciji biće reči o poboljšanju našeg algoritma uz uvođenje premisa koje postoje u stvarnosti, a koje smo do sada zanemarivali da bismo dali neke početne načine za rešavanje.

Kada smo govorili o proteinima, rekli smo da 20 aminokiselina najčešće učestvuje u njihovoj izgradnji i da su za nas, sa računarske tačke gledišta, proteini niske nad azbukom od 20 karaktera i da postoji još veliki broj aminokiselina nezavisno od izgradnje proteina u ćelijama živih bića. U gentskom kodu postoje kodovi samo za tih 20 aminokiselina, i u tabeli celobrojnih masa aminokiselina postoje mase samo za iste te aminokiseline. S obzirom da u ovom poglavlju razmatramo NRP peptide, peptide koji ne nastaju prema pravilima centralne dogme, onda ovi peptidi mogu da sadrže i neke nestandardne aminokiseline, one aminokiseline koje se ne nalaze među standardnih 20 aminokiselina. Na primer, tirocidin B sadrži nestandardnu aminokiselinu *Ornitin (Orn)*. Za Ornitin ne postoji nukleotidni triplet u okviru genetskog koda na osnovu koga se ova aminokiselina dobija i ne postoji celobrojna masa u tabeli celobrojnih masa za aminokiseline. S ozbirom na to, možemo da pretpostavimo da bilo koji ceo broj između 57 i 200 (koliko nam iznosi najmanja i najveća masa standardnih aminokiselina) može biti masa neke nestandardne aminokiseline. Ovako nešto može da izgleda kao grubo ograničenje, ali je eksperimentalno potvrđeno da većina masa svih mogućih aminokiselina pripada ovom intervalu.

Spektar u kome nismo ograničeni na tabelu od samo 18 celobrojnih masa, već uzimamo u obzir da bilo koji celi broj između 57 i 200 može da označava neku aminokiselinu, nazivamo **prošireni spektar**. Kada primenimo Leaderboard algoritam na prošireni spektar sa 10% lažnih i nedostajućih masa, peptid koji dobijemo $VKLFPWFN-98-65$ sadrži neke vrednosti za mase koje ne odgovaraju nijednoj aminokiselini. Pošto Leaderboard algoritam ovde ne daje ispravne vrednosti, moramo da primenimo jedan sasvim novi princip.

## 3.6 Spektralna konvolucija

Kod algoritma sa proširenim spektrom podrazumeva se da svi celi brojevi između 57 i 200 odgovaraju masama aminokiselina. To znači da razmatramo 144 ili više (znamo da jednoj masi može da odgovara više aminokiselina, a sa druge strane postoje vrednosti kojima ne odgovara nijedna) aminokiselina u koje spadaju i standardne i nestandardne aminokiseline. Želimo da smanjimo broj aminokiselina koje razmatramo.

Posmatrajmo eksperimentalni spektar za $NQEL$

0 99 113 114 128 227 257 299 355 356 370 371 484.

Mi znamo da je $Mass(E) = 129$ i vidimo da u spektru ne postoji ta vrednost. Sa druge strane, u spektru postoji $Mass(QE) = 257$ i $Mass(Q) = 128$. Razlika ove dve mase daje vrednost 129. Ova vrednost već deluje kao dobra vrednost za nedostajuću masu. U spektru postoji još ovakvih slučajeva. Recimo, $Mass(ELN) - Mass(LN) = 356 - 227 = 129$ i $Mass(NQEL) - Mass(LNQ) = 484 - 355 = 129$. Obe ove razlike ukazuju na masu od $E$ koja nedostaje.

Uvodimo tabelu koja se naziva **spektralna konvolucija**.

**Definicija 3.6.** ***Spektralna konvolucija*** *je tabela koja pokazuje apsolutnu vrednost razlike između svake dve mase u spektru.*

Primer spektralne konvolucije za spektar čije su lažne vrednosti označene sa ,,false" prikazan je na slici **??**.

| | "" | false | L | N | Q | LN | QE | false | LNQ | ELN | QEL | NQE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 99 | 113 | 114 | 128 | 227 | 257 | 299 | 355 | 356 | 370 | 371 |
| 0 | | | | | | | | | | | | |
| 99 | 99 | | | | | | | | | | | |
| 113 | 113 | 14 | | | | | | | | | | |
| 114 | 114 | 15 | 1 | | | | | | | | | |
| 128 | 128 | 29 | 15 | 14 | | | | | | | | |
| 227 | 227 | 128 | 114 | 113 | 99 | | | | | | | |
| 257 | 257 | 158 | 144 | 143 | 129 | 30 | | | | | | |
| 299 | 299 | 200 | 186 | 185 | 171 | 72 | 42 | | | | | |
| 355 | 355 | 256 | 242 | 241 | 227 | 128 | 98 | 56 | | | | |
| 356 | 356 | 257 | 243 | 242 | 228 | 129 | 99 | 57 | 1 | | | |
| 370 | 370 | 271 | 257 | 256 | 242 | 143 | 113 | 71 | 15 | 14 | | |
| 371 | 371 | 272 | 258 | 257 | 243 | 144 | 114 | 72 | 16 | 15 | 1 | |
| 484 | 484 | 385 | 371 | 370 | 356 | 257 | 227 | 185 | 129 | 128 | 114 | 113 |

Slika 3.8: Primer spektralne konvolucije.

Na preseku svake vrste i kolone u spektralnoj konvoluciji upisana je apsolutna vrednost razlike celobrojnih masa. Kako iskoristiti spektralnu konvoluciju? Tražimo vrednosti razlika koje se pojavljuju najveći broj puta, a da se nalaze između 57 i 200. Obojene vrednosti na slici **??** se pojavljuju veći broj puta. To su vrednosti 99, 113, 114, 128 i 129. Ove vrednosti odgovaraju masama aminokiselina, redom, $V, L, N, Q, E$. Od 5 najčešćih aminokiselina u konvoluciji 4 čine peptid $NQEL$.

Kako bi izgledao unapređeni algoritam za sekvenciranje ciklopeptida ukoliko uzmemo u obzir i nestandardne aminokiseline, odnosno proširenu tabelu celobrojnih masa aminokiselina? Pseudokod je dat u nastavku.

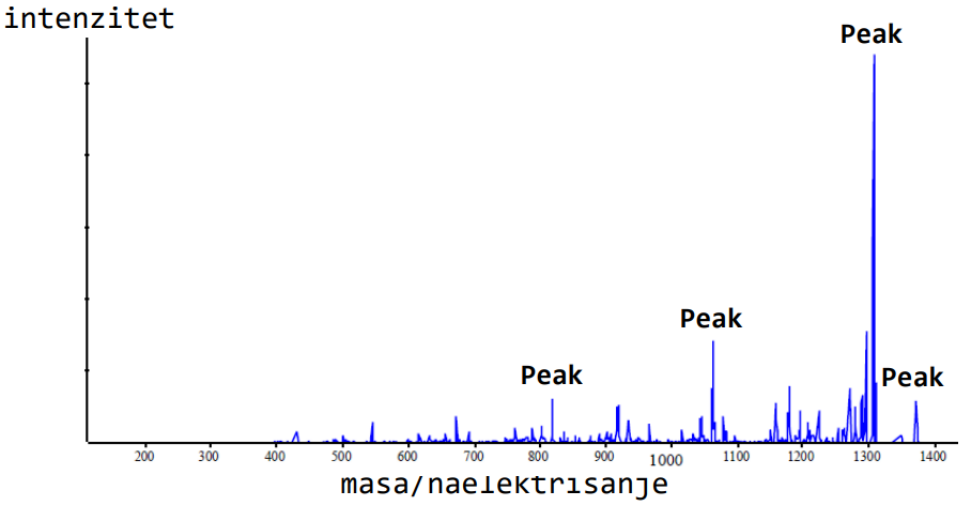
```
1 ConvolutionCyclopeptideSequencing(Spectrum, N, M)
2 begin
3     Formirati spektralnu konvoluciju spektra Spectrum.
4     Uzeti M najcescih elemenata u konvoluciji (izmedju 57 i 200).
5     Primeniti LeaderboardCyclopeptideSequencing, formirajuci peptide samo na
      ↪ osnovu ovih M celih brojeva.
6 end
```

Algoritam *ConvolutionCyclopeptideSequencing* daje tačan rezultat i za spektre sa šumom od 10% i za spektre sa šumom od 25%, što pokazuje da je spektralna konvolucija odgovorila na sve izazove koji su postavljeni.

## 3.7 Spektri u realnosti

Kao što znamo, realnost je obično drugačija. Neke poteškoće iz realnosti smo zanemarivali. Koje?

- $Spectrum_{25}$ je mnogo manje šumovit nego spektri dobijeni u praksi iz masenog spektrometra.
- Maseni spektrometar ne meri jednostavno fragmente podpeptida, već su postupci merenja mnogo komplikovaniji. Najpre se zaista vrši razbijanje datog peptida na fragmente. Zatim se oni sortiraju, korišćenjem elektromagnetnog polja, prema svojoj masi. Ono što maseni spektrometar meri jeste zapravo **odnos mase i naelektrisanja** za svaki fragment (znači nije baš masa) i određuje **intenzitet** (kao broj jona) u svakom odnosu mase i naelektrisanja. Šta to znači? To znači da kao izlaz iz masenog spektrometra ne dobijamo eksperimentalni spektar koji smo do sada imali prilike da vidimo, nego grafik intenziteta prema odnosu mase i naelektrisanja sa vrhovima na određenim mestima. Primer ovakvog grafika dat je na slici **??**. Na osnovu vrhova na grafiku, određivaćemo sam sastav peptida. Ovaj grafik se




Slika 3.9: Primer grafika intenziteta prema odnosu mase i naelektrisanja.

naziva **realni spektar**. Rekonstrukcija peptida na osnovu realnog spektra biće obrađena u poglavlju 11.

## 3.8 Zadaci sa vežbi

U nastavku će biti predstavljeni zadaci sa vežbi na kursu rađeni u programskom jeziku Python.

### 3.8.1 Linear Spectrum

```

# Formiranje linearnog spektra zadatog peptida
def linear_spectrum(peptide, amino_acid, amino_acid_mass):
    prefix_mass = [0]
    current_mass = 0
    for i in range(len(peptide)):
        for j in range(20):
            if amino_acid[j] == peptide[i]:
                prefix_mass.append(current_mass + amino_acid_mass[j])
                current_mass += amino_acid_mass[j]

    linear_spectrum = [0]
    for i in range(len(prefix_mass)):
        for j in range(i+1, len(prefix_mass)):
            linear_spectrum.append(prefix_mass[j] - prefix_mass[i])

    linear_spectrum.sort()
    return linear_spectrum


def main():

    # Lista aminokiselina
    amino_acid = ['G', 'A', 'S', 'P', 'V', 'T', 'C', 'I', 'L', 'N', 'D', 'K', '
    ↪ Q', 'E', 'M', 'H', 'F', 'R', 'Y', 'W']

    # Lista masa odgovarajucih aminokiselina
    amino_acid_mass = [57, 71, 87, 97, 99, 101, 103, 113, 113, 114, 115, 128,
    ↪ 128, 129, 131, 137, 147, 156, 163, 186]

    # Zadati peptid
    peptide = "NQEL"

    spectrum = linear_spectrum(peptide, amino_acid, amino_acid_mass)
    print(spectrum)

if __name__ == "__main__":
    main()
```

### 3.8.2 Cyclic Spectrum

```

# Formiranje ciklicnog spektra zadatog peptida
def cyclic_spectrum(peptide, amino_acid, amino_acid_mass):
    prefix_mass = [0]
    current_mass = 0
    for i in range(len(peptide)):
```

```
7           for j in range(20):
8               if amino_acid[j] == peptide[i]:
9                   prefix_mass.append(current_mass + amino_acid_mass[j])
10                  current_mass += amino_acid_mass[j]
11
12      peptide_mass = prefix_mass[-1]
13      cyclic_spectrum = [0]
14      for i in range(len(prefix_mass)):
15          for j in range(i+1, len(prefix_mass)):
16              cyclic_spectrum.append(prefix_mass[j] - prefix_mass[i])
17              if i > 0 and j < len(prefix_mass)-1:
18                  cyclic_spectrum.append(peptide_mass - (prefix_mass[j] -
    ↪ prefix_mass[i]))
19
20      cyclic_spectrum.sort()
21      return cyclic_spectrum
22
23
24  def main():
25      amino_acid = ['G', 'A', 'S', 'P', 'V', 'T', 'C', 'I', 'L', 'N', 'D', 'K', '
    ↪ Q', 'E', 'M', 'H', 'F', 'R', 'Y', 'W']
26      amino_acid_mass = [57, 71, 87, 97, 99, 101, 103, 113, 113, 114, 115, 128,
    ↪ 128, 129, 131, 137, 147, 156, 163, 186]
27
28      peptide = "NQE"
29
30      spectrum = cyclic_spectrum(peptide, amino_acid, amino_acid_mass)
31      print(spectrum)
32
33  if __name__ == "__main__":
34      main()
```

### 3.8.3 Cyclopeptide Sequencing

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

```
19                 cyclic_spectrum.append(peptide_mass - (prefix_mass[j] -
    ↪ prefix_mass[i]))
20
21     cyclic_spectrum.sort()
22     return cyclic_spectrum
23
24 # Prosirivanje liste peptida dodavanjem svih mogucih amino kiselina na kraj
    ↪ lanca
25 def expand(peptides, amino_acid):
26     extension = []
27
28     for peptide in peptides:
29         for aa in amino_acid:
30             extension.append(peptide + aa)
31
32     return extension
33
34 # Izracunavanje ukupne mase peptida kao sume svih aminokiselina u lancu
35 def mass(peptide, amino_acid, amino_acid_mass):
36     total_mass = 0
37
38     for i in range(len(peptide)):
39         for j in range(len(amino_acid)):
40             if peptide[i] == amino_acid[j]:
41                 total_mass += amino_acid_mass[j]
42
43     return total_mass
44
45 # Izdvajanje sume celog peptida iz spektra
46 def parent_mass(spectrum):
47     return spectrum[-1]
48
49 # Formiranje linearnog spektra
50 def linear_spectrum(peptide, amino_acid, amino_acid_mass):
51     prefix_mass = [0]
52     current_mass = 0
53     for i in range(len(peptide)):
54         for j in range(20):
55             if amino_acid[j] == peptide[i]:
56                 prefix_mass.append(current_mass + amino_acid_mass[j])
57                 current_mass += amino_acid_mass[j]
58
59     linear_spectrum = [0]
60     for i in range(len(prefix_mass)):
61         for j in range(i+1, len(prefix_mass)):
62             linear_spectrum.append(prefix_mass[j] - prefix_mass[i])
63
64     linear_spectrum.sort()
65     return linear_spectrum
66
67
68 # Provera da li je dati peptid konzistentan sa zadatim spektrom
69 def consistent(peptide, target_spectrum, amino_acid, amino_acid_mass):
```

```
70      peptide_linear_spectrum = linear_spectrum(peptide, amino_acid,
        ↪ amino_acid_mass)
71
72      for aa in peptide_linear_spectrum:
73          found = False
74          for aa_p in target_spectrum:
75              if aa_p == aa:
76                  found = True
77          if found == False:
78              return False
79
80      return True
81
82
83  # Sekvenciranje ciklopeptida
84  def cyclopeptide_sequencing(spectrum, amino_acid, amino_acid_mass):
85      peptides = ['']
86      i = 1;
87      while len(peptides) > 0:
88          next_peptides = []
89          peptides = expand(peptides, amino_acid)
90          next_peptides = copy.copy(peptides)
91          for peptide in peptides:
92              if mass(peptide, amino_acid, amino_acid_mass) == parent_mass(
        ↪ spectrum):
93                  if cyclic_spectrum(peptide, amino_acid, amino_acid_mass) ==
        ↪ spectrum:
94                      print(peptide)
95                  next_peptides.remove(peptide)
96              elif not consistent(peptide, spectrum, amino_acid, amino_acid_mass)
        ↪ :
97                  next_peptides.remove(peptide)
98          peptides = next_peptides
99
100
101 def main():
102     amino_acid = ['G', 'A', 'S', 'P', 'V', 'T', 'C', 'I', 'L', 'N', 'D', 'K', '
        ↪ Q', 'E', 'M', 'H', 'F', 'R', 'Y', 'W']
103     amino_acid_mass = [57, 71, 87, 97, 99, 101, 103, 113, 113, 114, 115, 128,
        ↪ 128, 129, 131, 137, 147, 156, 163, 186]
104
105     peptide = "SPQR"
106
107     spectrum = cyclic_spectrum(peptide, amino_acid, amino_acid_mass)
108
109     cyclopeptide_sequencing(spectrum, amino_acid, amino_acid_mass)
110
111 if __name__ == "__main__":
112     main()
```

### 3.8.4 Leaderboard Cyclopeptide Sequencing

```
1  import copy
```

```
2
3  # Formiranje ciklicnog spektra peptida
4  def cyclic_spectrum(peptide, amino_acid, amino_acid_mass):
5      prefix_mass = [0]
6      current_mass = 0
7      for i in range(len(peptide)):
8          for j in range(20):
9              if amino_acid[j] == peptide[i]:
10                 prefix_mass.append(current_mass + amino_acid_mass[j])
11                 current_mass += amino_acid_mass[j]
12
13     peptide_mass = prefix_mass[-1]
14     cyclic_spectrum = [0]
15     for i in range(len(prefix_mass)):
16         for j in range(i+1, len(prefix_mass)):
17             cyclic_spectrum.append(prefix_mass[j] - prefix_mass[i])
18             if i > 0 and j < len(prefix_mass)-1:
19                 cyclic_spectrum.append(peptide_mass - (prefix_mass[j] -
    ↪ prefix_mass[i]))
20
21     cyclic_spectrum.sort()
22     return cyclic_spectrum
23
24 # Prosirivanje liste peptida dodavanjem svih mogucih amino kiselina na kraj
    ↪ lanca
25 def expand(peptides, amino_acid):
26     extension = []
27
28     for peptide in peptides:
29         for aa in amino_acid:
30             extension.append(peptide + aa)
31
32     return extension
33
34 # Izracunavanje ukupne mase peptida kao sume svih aminokiselina u lancu
35 def mass(peptide, amino_acid, amino_acid_mass):
36     total_mass = 0
37
38     for i in range(len(peptide)):
39         for j in range(len(amino_acid)):
40             if peptide[i] == amino_acid[j]:
41                 total_mass += amino_acid_mass[j]
42
43     return total_mass
44
45 # Izdvajanje sume celog peptida iz spektra
46 def parent_mass(spectrum):
47     return spectrum[-1]
48
49 # Formiranje linearnog spektra
50 def linear_spectrum(peptide, amino_acid, amino_acid_mass):
51     prefix_mass = [0]
52     current_mass = 0
```

```
53      for i in range(len(peptide)):
54          for j in range(20):
55              if amino_acid[j] == peptide[i]:
56                  prefix_mass.append(current_mass + amino_acid_mass[j])
57                  current_mass += amino_acid_mass[j]
58
59      linear_spectrum = [0]
60      for i in range(len(prefix_mass)):
61          for j in range(i+1, len(prefix_mass)):
62              linear_spectrum.append(prefix_mass[j] - prefix_mass[i])
63
64      linear_spectrum.sort()
65      return linear_spectrum
66
67
68  # Provera da li je dati peptid konzistentan sa zadatim spektrom
69  def consistent(peptide, target_spectrum, amino_acid, amino_acid_mass):
70      peptide_linear_spectrum = linear_spectrum(peptide, amino_acid,
        ↪ amino_acid_mass)
71
72      for aa in peptide_linear_spectrum:
73          found = False
74          for aa_p in target_spectrum:
75              if aa_p == aa:
76                  found = True
77          if found == False:
78              return False
79
80      return True
81
82
83  def score(peptide, spectrum_2, amino_acid, amino_acid_mass):
84      p1 = 0
85      p2 = 0
86      score = 0
87
88      spectrum_1 = cyclic_spectrum(peptide, amino_acid, amino_acid_mass)
89
90      while p1 < len(spectrum_1) and p2 < len(spectrum_2):
91          if spectrum_1[p1] == spectrum_2[p2]:
92              score += 1
93              p1 += 1
94              p2 += 1
95          elif spectrum_1[p1] < spectrum_2[p2]:
96              p1 += 1
97          else:
98              p2 += 1
99
100     return score
101
102 def linear_score(peptide, spectrum_2, amino_acid, amino_acid_mass):
103     p1 = 0
104     p2 = 0
```

```
105     score = 0
106
107     spectrum_1 = linear_spectrum(peptide, amino_acid, amino_acid_mass)
108
109     while p1 < len(spectrum_1) and p2 < len(spectrum_2):
110         if spectrum_1[p1] == spectrum_2[p2]:
111             score += 1
112             p1 += 1
113             p2 += 1
114         elif spectrum_1[p1] < spectrum_2[p2]:
115             p1 += 1
116         else:
117             p2 += 1
118
119     return score
120
121 # Sekvenciranje ciklopeptida
122 def leaderboard_cyclopeptide_sequencing(spectrum, N, amino_acid,
    ↪ amino_acid_mass):
123     leaderboard = ['']
124     leader_peptide = ''
125     while len(leaderboard) > 0:
126         next_peptides = []
127         leaderboard = expand(leaderboard, amino_acid)
128         next_leaderboard = copy.copy(leaderboard)
129         for peptide in leaderboard:
130             if mass(peptide, amino_acid, amino_acid_mass) == parent_mass(
    ↪ spectrum):
131                 if score(peptide, spectrum, amino_acid, amino_acid_mass) >
    ↪ score(leader_peptide, spectrum, amino_acid, amino_acid_mass):
132                     leader_peptide = peptide
133             elif mass(peptide, amino_acid, amino_acid_mass) > parent_mass(
    ↪ spectrum):
134                 next_leaderboard.remove(peptide)
135         leaderboard = trim(next_leaderboard, spectrum, N, amino_acid,
    ↪ amino_acid_mass)
136     return leader_peptide
137
138
139 def trim(leaderboard, spectrum, N, amino_acid, amino_acid_mass):
140     linear_scores = []
141     for j in range(len(leaderboard)):
142         peptide = leaderboard[j]
143         linear_scores.append(linear_score(peptide, spectrum, amino_acid,
    ↪ amino_acid_mass))
144
145     leaderboard_zipped = list(zip(linear_scores, leaderboard))
146     leaderboard_zipped.sort(reverse=True)
147
148     leaderboard = [el[1] for el in leaderboard_zipped]
149     for j in range(N, len(leaderboard_zipped)):
150         if leaderboard_zipped[j][0] < leaderboard_zipped[N-1][0]:
151             leaderboard = [el[1] for el in leaderboard_zipped[:j]]
```

```
152             return leaderboard
153     return leaderboard
154
155
156
157
158 def main():
159     amino_acid = ['G', 'A', 'S', 'P', 'V', 'T', 'C', 'I', 'L', 'N', 'D', 'K', '
        ↪ Q', 'E', 'M', 'H', 'F', 'R', 'Y', 'W']
160     amino_acid_mass = [57, 71, 87, 97, 99, 101, 103, 113, 113, 114, 115, 128,
        ↪ 128, 129, 131, 137, 147, 156, 163, 186]
161
162     peptide = "SPQR"
163
164     spectrum = cyclic_spectrum(peptide, amino_acid, amino_acid_mass)
165
166     print(leaderboard_cyclopeptide_sequencing(spectrum, 10, amino_acid,
        ↪ amino_acid_mass))
167
168 if __name__ == "__main__":
169     main()
```