Bioinformatika

25. mart 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

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.

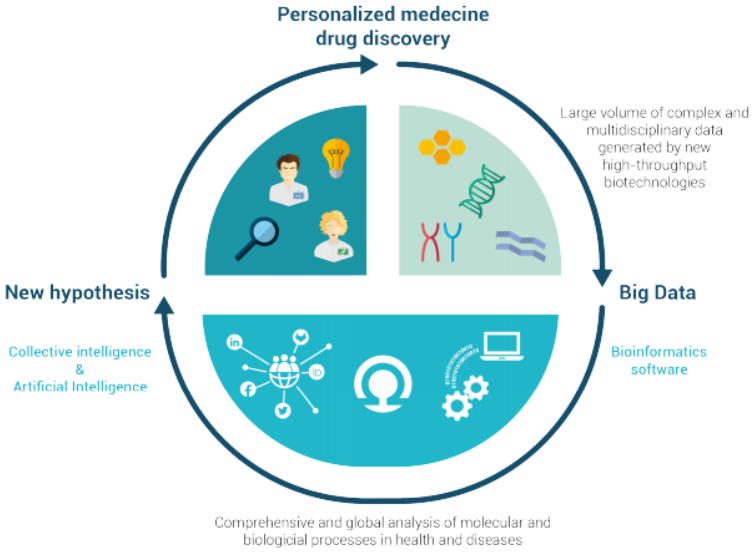
Bioinformatika ima široku primenu i njene primene rastu zajedno sa razvojem discipline. Kao što možemo videti na slici ispod, primena bioinformatike se može sagledati kroz personalizovanu medicinu. Naime, na osnovu prikupljene veće količine podataka i njihove analize, uz pomoć različitih računarskih 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.

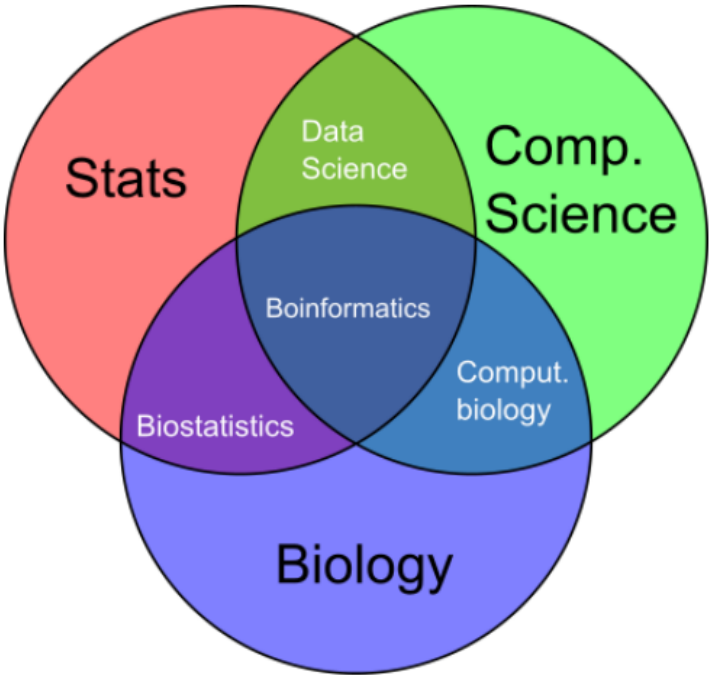
DNK se sastoji iz dva polimerna lanca koji imaju antiparalelnu orijentaciju, i svaki od njih je sastavljen od azotnih baza:

- 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.

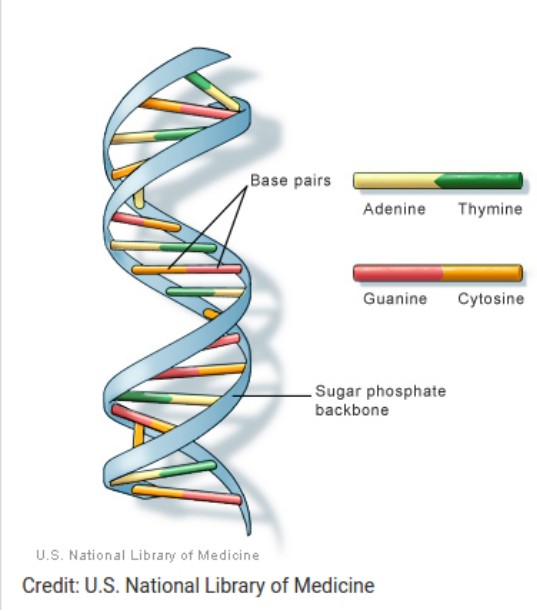
Dzejms Votson (engl. *James Watson*) i Fransis Krik (engl. *Fransis Crick*) su 1953. godine napisali rad u kome su primetili da postoji mehanizam za kopiranje genetskog materijala. Oni su uočili da se lanci roditeljskog DNK molekula odvijaju tokom replikacije i da se, potom, svaki lanac ponaša kao uzorak 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.

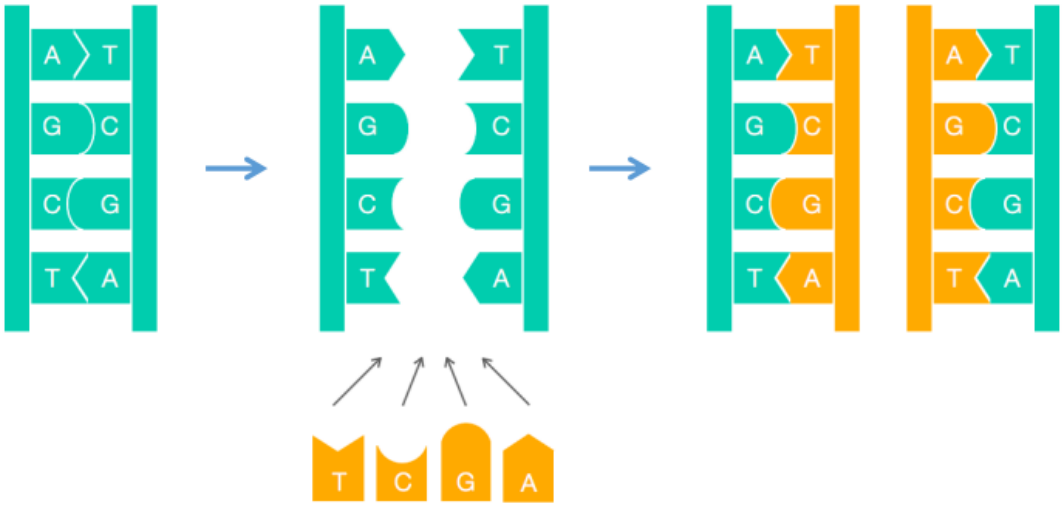
Poznato je da DNKA utiče na početak replikacije. *DNKA* je protein koji se vezuje na kratki segment unutar oriC, poznatiji kao **DNKA boks**. Ona predstavlja poruku unutar sekvence DNK koja govori proteinu DNKA da se veže baš tu. Postavlja se pitanje kao pronaći taj region bez prethodnog poznavanja izgleda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

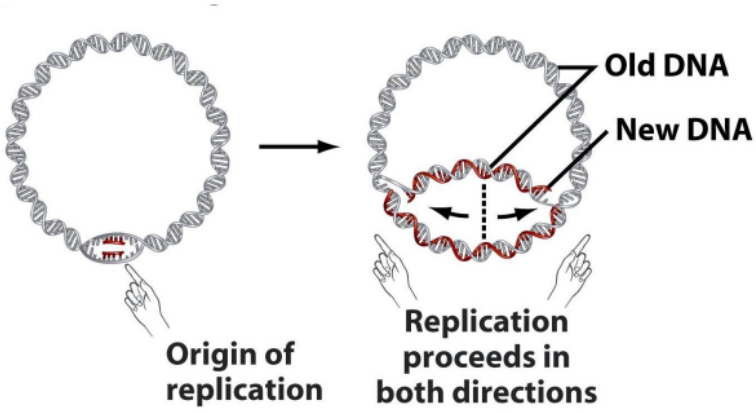
```
53++!305))6*;4826)4+.)4+);806*;48!8‘6 0))85;]8*:+*8!83(88)5*!;46(;88*96*?;8 )*+(;485);5*!2
:*+(;4956*2(5*4)8‘8*;40 69285);)6!8)4++;1(+9;48081;8:8+1;48!8 5;4)485
!528806*81(+9;48;(88;4(+?34;48 )4+;161;:188;+?;
```

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.

```
PATTERNCOUNT(Text, Pattern)
    count = 0
    for i = 0 to |Text| - |Pattern|
        if Text(i,|Pattern|) = Pattern
            count = count + 1
    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).

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?

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

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. Verovatnoća da postoji 9-gram koji se pojavljuje 3 ili više puta u proizvoljno generisanom DNK stringu dužine 500 je 1/1300. Uočimo da postoje četiri različita 9-grama koji se ponavljaju tri ili više puta u ovom regionu, to su: ATGATCAAG, CTTGATCAT, TCTTGATCA i CTCTTGATC.

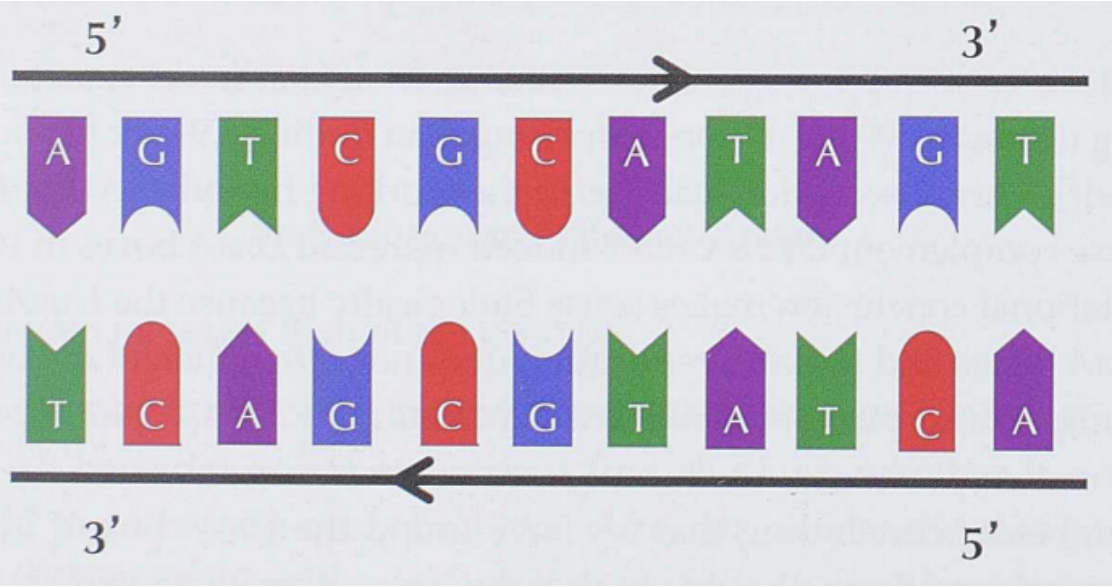
Slika 1.7: Prikaz 9-grama ATGATCAAG i njegovog komplementa u *oriC* regionu Vibrio cholerae

atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaacctgagtgg
atgacatcaagataggtcgttgtatctccttcctctcgtactctcatgaccacggaaag**ATGATCAAG**
agaggatgatttcttggccatatcgcaatgaatacttgtgacttgtgcttccaattgacatcttcagc
gccatattgcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgttctgttt
atcttgttttgactgagacttgttaggatagacggtttttcatcactgactagccaaagccttactct
gcctgacatcgaccgtaaattgataatgaatttacatgcttccgcgacgatttacct**CTTGATCAT**cg
atccgattgaagatcttcaattgttaattctcttgcctcgactcatagccatgatgagct**CTTGATCA**
Tgtttccttaaccctctatttttacggaaga**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.

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



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.

Slika 1.9: Prikaz *oriC* regiona Thermotoga petrophila

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

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.

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.

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

```
aactctatacctccttttttgtcgaatttgtgtgatttatagagaaaatcttattaactgaaactaa
aatggtaggtttGGTGGTAGGttttgtgtacattttgtagtatctgatttttaattacataccgta
tattgtattaaattgacgaacaattgcatggaattgaatatatgcaaaacaaaCCTACCACCaaac
tctgtattgaccattttaggacaacttcagGGTGGTAGGtttctgaagctctcatcaatagactat
tttagtctttacaaacaatattaccgttcagattcaagattctacaacgctgttttaatgggcgtt
gcagaaaacttaccacctaaaatccagtatccaagccgatttcagagaaacctaccacttacctac
cacttaCCTACCACCcgggtggtaagttgcagacattattaaaaacctcatcagaagcttgttcaa
aaatttcaatactcgaaaCCTACCACCtgcgtcccctattatttactactactaataatagcagta
taattgatctgaaaagaggtggtaaaaaa
```

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

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?

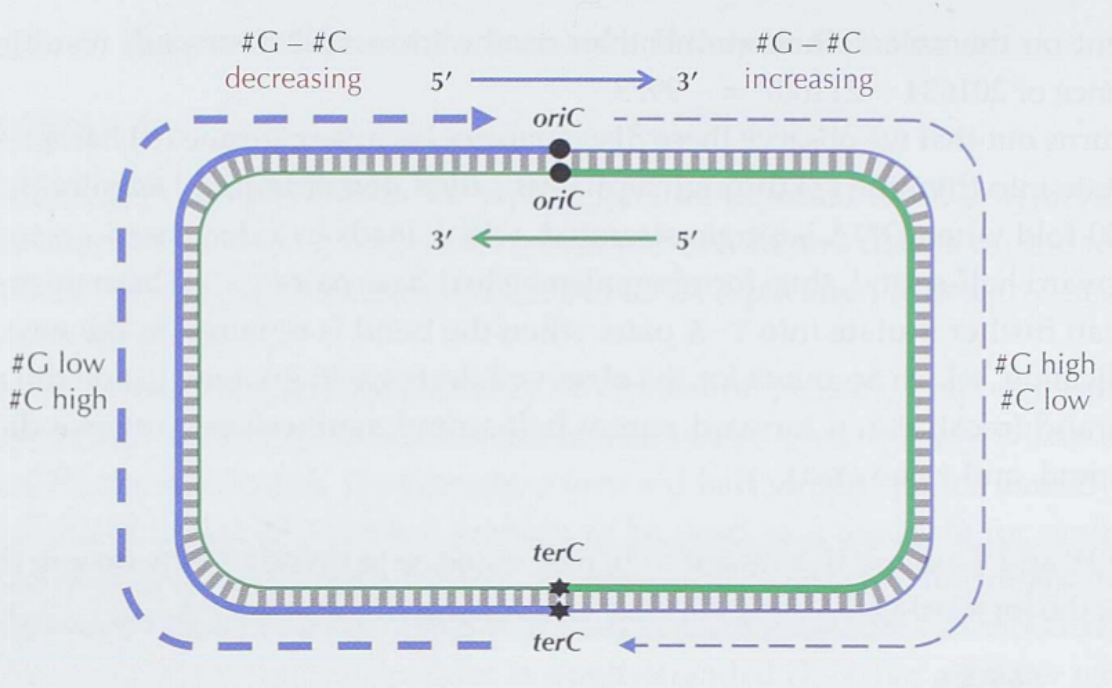
Iskrivljeni dijagrami

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.

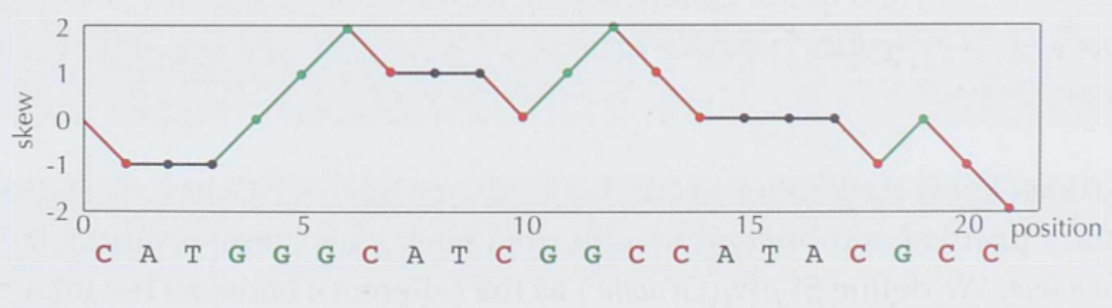
Posmatrajmo iskrivljeni dijagram bakterije Ešerihija Koli. Lako uočavamo minimalnu vrednost skew dijagrama.

Minimalna vrednost iz iskrivljenog dijagrama ukazuje baš na ovaj region:

Slika 1.11: Prikaz kretanja.



Slika 1.12: Iskrivljeni dijagram genoma Genome = CATGGGCATCGGCCATACGCC.

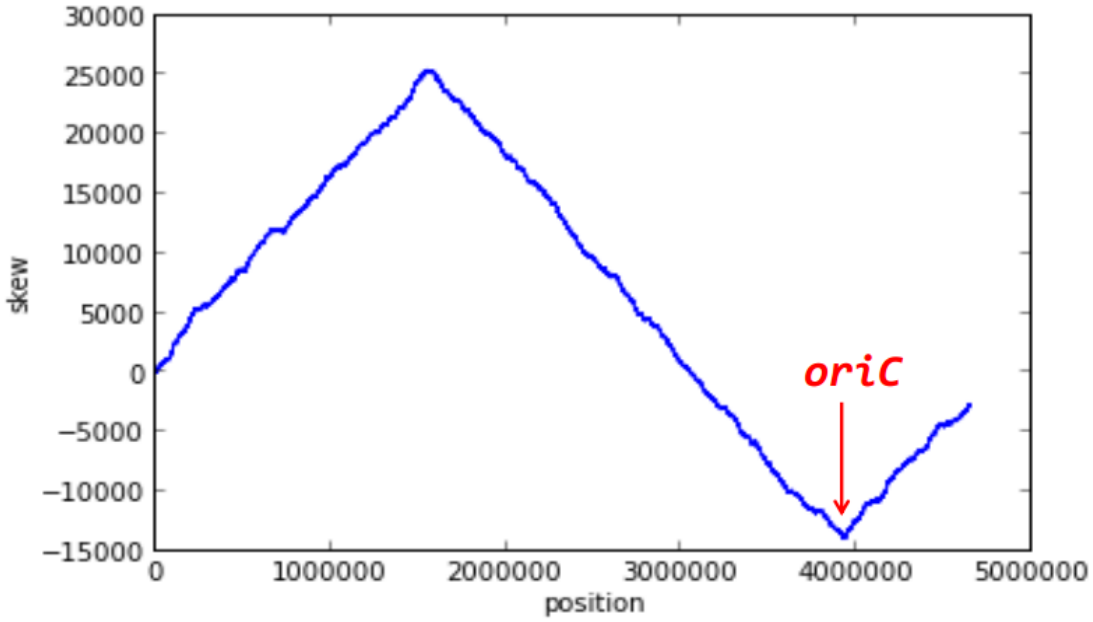


Slika 1.14: Region na koji pokazuje minimalna vrednost iskrivljenog dijagrama Ešerihije koli.

```
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

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

def frequent_words(text, k, min_count):
    frequent_patterns = set([])
    count = []
    n = len(text)-k
    for i in range(n):
        # Izvuci podnisku koji pocinje na $i$-toj poziciji i ima $k$ karaktera
        pattern = text[i:i+k]
        count.append(pattern_count(text, pattern))
    max_count = max(count)
    if max_count < min_count:
        return []
    for i in range(n):
        if count[i] == max_count:
            frequent_patterns.add(text[i:i+k])
    return frequent_patterns

def main():
    print(frequent_words('agctagatgctagctagctgatcgagctgatgcaggcagtgctagc', 4,
    ↪ 2))

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

1.3.2 Faster FrequentWords

```
#faster frequent_words
def pattern_to_number(pattern):
    if len(pattern) == 0:
        return 0
    last = pattern[-1]
    prefix = pattern[:-1]
    return 4 * pattern_to_number(prefix) + symbol_to_number(last)

def number_to_pattern(n, k):
    if k == 1:
        return number_to_symbol(n)
    prefixIndex = n // 4 #celobrojno deljenje
    r = n  %  4
```

```
    symbol = number_to_symbol(r)
    prefix  = number_to_pattern(prefixIndex, k-1)
    return prefix + symbol

def symbol_to_number(c):
    pairs = {
        'a' : 0,
        't' : 1,
        'c' : 2,
        'g' : 3
    }
    return pairs[c]

def number_to_symbol(n):
    pairs = {
        0 : 'a',
        1 : 't',
        2 : 'c',
        3 : 'g'
    }
    return pairs[n]

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

def computing_frequencies(text,k):
    frequency_array = [0 for i in range(4**k)] #4 ^ k
    for i in range(len(text) - k):
        pattern = text[i:i+k]
        j = pattern_to_number(pattern)
        frequency_array[j] += 1
    return frequency_array

def faster_frequent_words(text, k, min_count):
    frequent_patterns = set([])
    frequency_array = computing_frequencies(text, k)
    max_count = max(frequency_array)
    if max_count < min_count:
        return []
    for i in range(4**k):
        if frequency_array[i] == max_count:
            pattern = number_to_pattern(i, k)
            frequent_patterns.add(pattern)
    return frequent_patterns

def main():
    print(faster_frequent_words('agctagatgctagctagctgatcgagctgatgcaggcagtgctagc
    ↪ ', 4, 2))
```

```
    #print(number_to_pattern(pattern_to_number('ta'),2))

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

1.3.3 Skew Diagram

```
#GC-skew
import matplotlib.pyplot as plt

def draw_skew(skew):
    x = [i for i in range(len(skew))]
    ax = plt.subplot()
    ax.plot(x, skew)
    plt.show()

def calculate_skew(text):
    skew = [0 for c in text]
    last = 0
    for i in range(0, len(text)):
        if text[i] == 'g':
            skew[i] = last + 1
        elif text[i] == 'c':
            skew[i] = last - 1
        else:
            skew[i] = last
        last = skew[i]
    return skew


def main():
    text = "catgggcatcggccatacgcc"
    print(calculate_skew(text))
    draw_skew(calculate_skew(text))

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

1.3.4 FrequentWords With Mismatches

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

    return pairs[c]

# Prevodjenje nukleotida u brojeve
def number_to_symbol(n):
    pairs = {
```

```
        0 : 'a',
        1 : 't',
        2 : 'c',
        3 : 'g'
    }

    return pairs[n]

# Prevodjenje broja u odgovarajucu nukleotidnu sekvencu
def number_to_pattern(n, k):
    if k == 1:
        return number_to_symbol(n)

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

    return prefix_pattern + c



# Prevodjenje nukleotidne sekvence u odgovarajuci broj
def pattern_to_number(pattern):
    if len(pattern) == 0:
        return 0

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

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



# Hamingova distanca, broj pozicija karaktera na kojima se tekstovi 1 i 2
    ↪ razlikuju,
# podrazumeva se da je duzina obe niske jednaka
def hamming_distance(text1, text2):
    distance = 0

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

    return distance



# Brojanje pojavljivanja podsekvenci u tekstu koje se od uzorka razlikuju na
    ↪ najvise d pozicija
def approximate_pattern_count(text, pattern, d):
    count = 0
```

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

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

    return count


# Pronalazenje svih niski susednih zadatom uzorku sa razlikama na najvise d
    ↪ pozicija
def neighbors(pattern, d):
    if d == 0: # Ako je dozvoljena greska jednaka nuli onda je samo uzorak svoj
    ↪  sused bez gresaka
        return set([pattern])

    # Izlaz iz rekurzije:
    if len(pattern) == 1: # Ako je duzina uzorka jednaka 1, a dozvoljena greska
    ↪  je veca od nule, onda je moguce iskoristiti bilo koji karakter
        return set(['a', 't', 'c', 'g'])

    neighborhood = set([])

    suffix_neighbors = neighbors(pattern[1:], d) # Pronalaze se svi susedi
    ↪ duzine n-1

    for text in suffix_neighbors:
        if hamming_distance(pattern[1:], text) < d: # Ako se sused razlikuje na
    ↪  manje od d pozicija od podniske uzorka bez prvog karaktera
            for x in ['a','t','c','g']:
                neighborhood.add(x + text) # Moguce je dodati bilo koji
    ↪ karakter na pocetak suseda i time dobiti najvise d razlika
        else: # U suprotnom, sused se vec razlikuje na d pozicija od uzorka pa
    ↪ je dozvoljeno samo dodavanje ispravnog karaktera kako se
            neighborhood.add(pattern[0] + text) # razlika ne bi povecala preko
    ↪ d

    return neighborhood



def frequent_words_with_mismatches(text, k, d):
    frequent_patterns = set([])

    close = [0 for i in range(4**k)] # Kandidati za proveru
    frequency_array = [0 for i in range(4**k)]

    # 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)
    for i in range(len(text) - k):
        neighborhood = neighbors(text[i:i+k], d) # Pronalaze se kandidati
```

```
        for pattern in neighborhood:
            index = pattern_to_number(pattern) # Svaka kandidat sekvenca se
    ↪ prevodi u svoj odgovarajuci indeks
            close[index] = 1 # i evidentira

    # Za svaku kandidat sekvencu (koja je do sada pronadjena u tekstu sa
    ↪ greskom d), broji se pojavljivanje njenih d-suseda
    for i in range(4**k):
        if close[i] == 1:
                pattern = number_to_pattern(i, k)
                frequency_array[i] = approximate_pattern_count(text, pattern, d
    ↪ )

    max_count = max(frequency_array)

    # Pronalaze se one sekvence duzine k cija su pojavljivanja najzastupljenija
    for i in range(4**k):
        if frequency_array[i] == max_count:
            pattern = number_to_pattern(i, k)
            frequent_patterns.add(pattern)

    return frequent_patterns



def main():
    print(frequent_words_with_mismatches('
    ↪ tgactatcatcgtagtatcgatgtgcacacacgtgcgcgcgcgcgccctgtacatgatc', 5, 2))

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

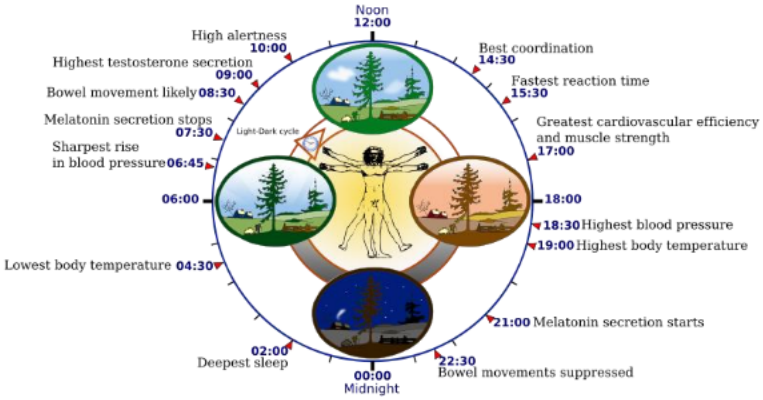
Glava 2

Koji DNK šabloni igraju ulogu molekularnog sata?

2.1 Biološki problem

Bioritam svih živih bića kotroliše "unutrašnji časovnik"koji još zovemo i cirkadijalni. Ljudi koji često putuju avionom na drugi kraj sveta mogu to da osete kada pokušaju da zaspu nakon promene nekoliko vremenskih zona. Kao i svaki sat, i ovaj može da se pokvari, što rezultuje genetskom bolešću pod nazivom sindrom odložene faze spavanja. Njegova osnova je na molekularnom nivou.

Slika 2.1: Cirkadijalni ritam



Naučnici su se pitali kako ćelije znaju kada treba da uspore ili ubrzaju proizvodnju određenih proteina. Ranih sedamdesetih, Ron Konopka i Sejmor Benzer su napravili prve korake ka rešavanju ove misterije. Do danas je otkriveno mnogo cirkadijalnih gena koji koordiniraju ponašanje stotine drugih gena.

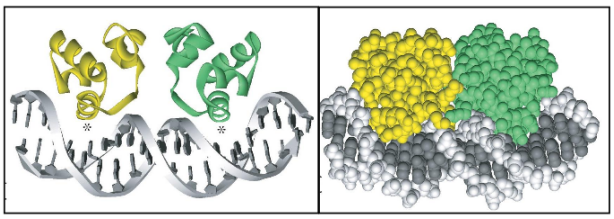
Kod čoveka, cirkadijalni ritam je promenljiv, tj. varira od osobe do osobe. Mi ćemo se u daljem tekstu fokusirati na biljke, jer je kod njih cirkadijalni ritam pitanje života i smrti, stoga ne sme biti promenljiv. Geni biljaka moraju znati kada sunce izlazi i zalazi kako bi znali kada treba vršiti fotosintezu, jer je onda od krucijalne važnosti za život biljke, a usko povezana sa količinom sunčeve svetlosti.

Ispostavlja se da svaka ćelija biljke čuva podatak o tome da li je dan ili noć nezavisno od drugih ćelija, kao i da su samo tri gena odgovorna za upravljanje satom. Oni kodiraju regulatorne proteine (transkripcione faktore)- to su LCY, CCA1 i TOC1. Spoljašnji faktori, kao što

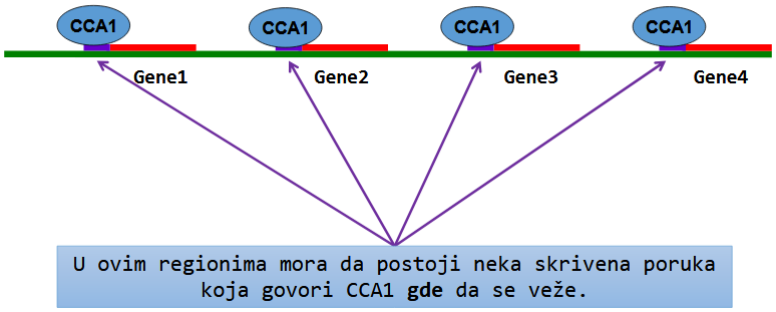
Slika 2.2: Cirkadijalni ritam biljke

je količina sunčeve svetlosti, kontrolišu regulatorne gene i regulatorne proteine kako bi organizmi prilagodili svoju gensku ekspresiju, odnosno da li će se protein sintetisati u to vreme ili ne. Dakle, svaki metabolički proces je regulisan cirkadijalnim časovnikom kojim upravljaju regulatorni proteini, odlučujući kada će se koji protein u ćeliji biljke sintetisati. Regulatorni proteini upravljaju genskom ekspresijom tako što se vežu za manji skup nukleotida u DNK koji nazivamo **regulatorni motiv**. Naš zadatak je da ih pronađemo.

Slika 2.3: Regulatorni proteini na delu



Slika 2.4: Mesta vezivanja regulatornih proteina



2.2 Informatički problem

Kako bismo informatički rešili problem pronalaženja regulatornih motiva u DNK, moramo predstaviti sve gorepomenute biološke pojmove na način koji bi informatičar i njegov računar mogli razumeti i obraditi.

- Iz ove tačke gledišta, DNK predstavlja nisku karaktera nad azbukom: A, G, T, C.
- Kodirajuće sekcije DNK (one koje se prepisuju i prevode u proteine) za nas će biti podniske niske DNK.
- Regulatorni motiv je šablon koji se pojavljuje tačnoo jednom u svakoj kodirajućoj sekciji DNK.

Hajde da definišemo naš problem na informatički način:

Informatička definicija problema
Ulaz: N niski koje predstavljaju kodirajuće sekvence DNK.
Izlaz: Podniska dužine k koja predstavlja regulatorni motiv (skrivenu poruku, mesto vezivanja regulatornih proteina).

Ovaj problem nas podseća na problem pronalaženja OriC-a, koji smo rešavali svodeći ga na pronalaženje čestih reči u tekstu koristeći algoritam FrequentWords, detaljno opisan u prethodnom poglavlju. Da bismo taj algoritam primenili ovde, neophodno je da od niza niski dobijemo konkatenacijom jednu veliku nisku, na koju ćemo primeniti ovaj algoritam.

Slika 2.5: DNK niska sa naznačenim regulatornim motivima

atgaccgggatactgatAAAAAAAAGGGGGGGggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
acccctatttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaacttttccgaataAAAAAAAAGGGGGGGa
tgagtatccctgggatgacttAAAAAAAAGGGGGGGtgctctcccgatttttgaatatgtaggatcattcgccagggtccga
gctgagaattggatgAAAAAAAAGGGGGGGtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
tccctttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatAAAAAAAAGGGGGGGcttatag
gtcaatcatgttcttgtgaatggatttAAAAAAAAGGGGGGGgaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
cggttttggcccttgttagaggcccccgtAAAAAAAAGGGGGGGcaattatgagagagctaatctatcgcgtgcgtgttcat
aacttgagttAAAAAAAAGGGGGGGctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatAAAAAAAAGGGGGGGaccgaaagggaag
ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttAAAAAAAAGGGGGGGa

Međutim, ovaj algoritam nije primenljiv ako motivi mutiraju. Možemo pokušati da primenimo algoitam FrequentWordsWithMissmatches. To bi nas dovelo do tačnog rešenja, ali nakon previše vremena. Naime, kada smo nalazili OriC, tražili smo uzorke dužine 9 karaktera (DnaA box je bio te dužine). U ovom slučaju, tražimo motive koji su najčešće dužine 15 karaktera, pa nam ovaj algoritam ne radi dovoljno brzo. Dakle, potrebna nam je nova ideja.

2.3 Problem ubačenog motiva

Najpre ćemo definisati još par pojmova.

- Mutirani šablon predstavlja šablon u kome se na nekim mestima može pojaviti mutacija, odnosno odstupanje od početne niske.
- (k, d) motiv je k-gram koji se pojavljuje u svakoj sekvenci sa najviše d razlika.
- Kanonski motiv predstavlja motiv koji tražimo (bez uticaja mutacija).
- Instance su mutirani motivi - oni koji se pojavljuju u niskama sa najviše d grešaka, odnosno razlika u odnosu na kanonski motiv.

Sada možemo definisati problem.

Problem ubačenog motiva: Pronalaženje (k, d) motiva u skupu niski
Ulaz: Skup niski Dna i celi brojevi k (dužina motiva) i d (maksimalni broj razlika).
Izlaz: Svi (k, d) motivi u skupu Dna.

Najzad, prikažimo nekoliko rešenja datog problema.

2.3.1 Enumeracija motiva

Početna ideja je zasnovana na gruboj sili - za svaki k-gram ćemo ispitati da li je (k, d) motiv za dati skup niski. Dakle, trebalo bi generisati 4^k kombinacija i za svaku ispitati da li je (k, d) motiv. Postavlja se pitanje da li je potrebno ispitati svih 4^k kandidata. Ispostavlja se da nije. To nas dovodi do bolje ideje: Proverićemo samo one kandidate koji se uopšte pojavljuju u nekoj niski iz Dna, a kao instance tražićemo samo one koji se od kandidata razlikuju na najviše d pozicija.

Slika 2.6: Enumeracija motiva

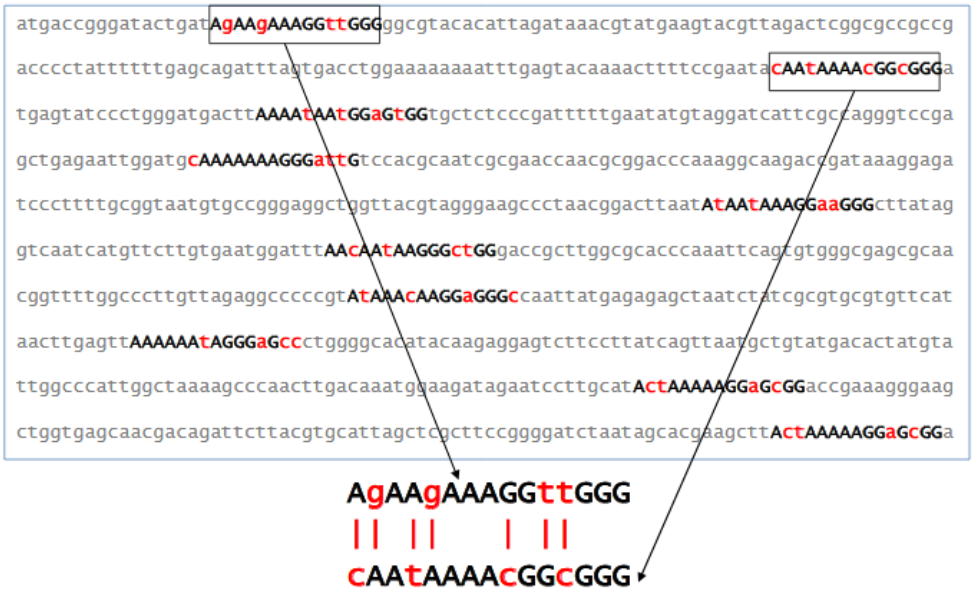
```
MotifEnumeration(Dna, k, d)
for each k-mer a in Dna
   •  generate all possible k-mers a' differing from a
      by at most d mutations
   •  for each such k-mer a'
         if a' is a (k,d)-mer in each sequence in Dna
             output a'
```

Ovaj algoritam je suviše spor kada su k i/ili d veliki brojevi.

2.3.2 Najsličniji k-grami u parovima niski

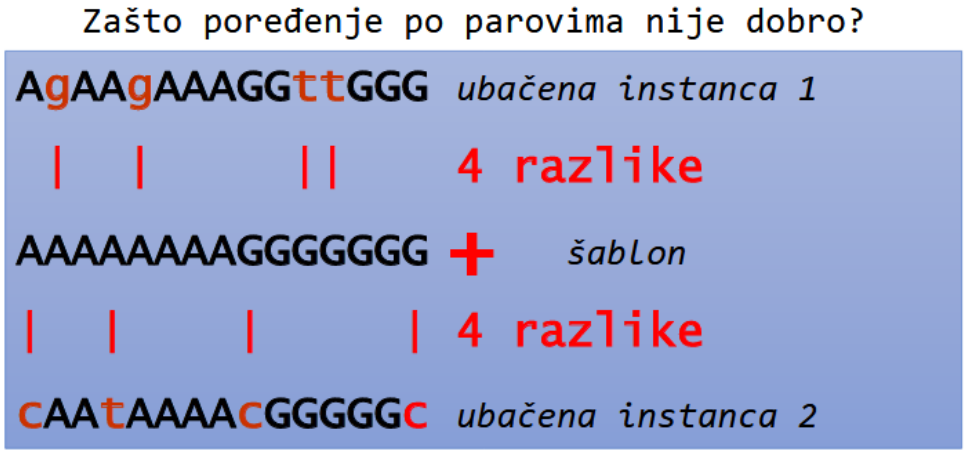
Kako bi ubrzali algoritam, možemo probati da poredimo parove niski iz Dna. Ideja je da uočimo dva najsličnija k-grama u dve niske iz Dna, od njih napravimo kanonski, i za njega proveravamo da li je (k, d) motiv.

Slika 2.7: Najsličniji k-grami u parovima niski - primer



Problem sa ovim pristupom leži u tome što ove dve početne instance mogu imati i do 2d razlika među sobom (zahtevali smo d razlika samo u odnosu na kanonski motiv koji smo od njih pravili).

Slika 2.8: Najsličniji k-grami u parovima niski - problem



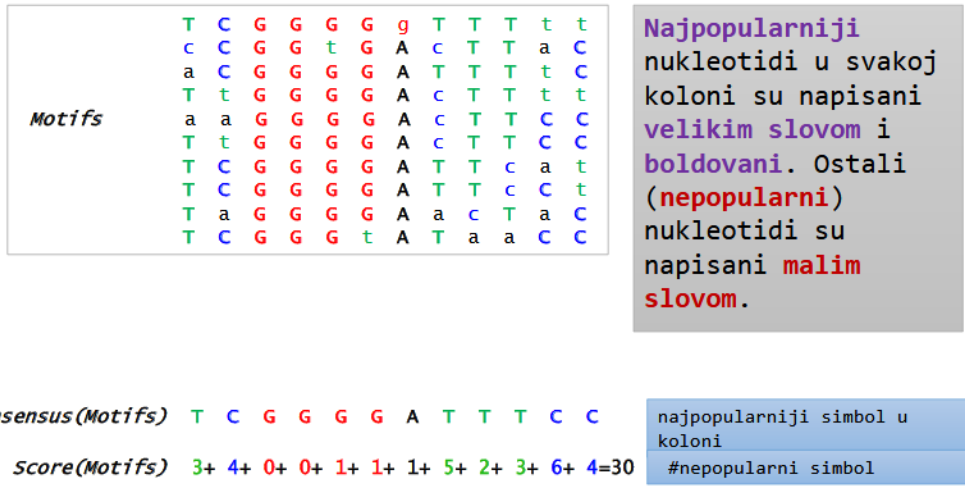
Primer koji pokazuje koliko je ovo zaista loše rešenje je eksperiment rađem mad 10 slučajno generisanih niski iz Dna dužine 600, sa ubačenim (15, 4) motivom. Pristupom pronalaženja parova niski iz Dna, pronađeno je nekoliko hiljada parova k-grama koji su se razlikovali na manje od 8 pozicija. Ovo sigurno nisu skrivene poruke, jer ih je previše. Potrebno nam je da ih nekako rangiramo. Dakle, potreban nam je novi pristup.

2.3.3 Matrice motiva

Najpre moramo uvesti nekoliko pojmova. Pretpostavićemo da imamo neku kolekciju motiva, stavićemo ih u matricu i definisati sledeće:

- Najpopularniji nukleotid u nekoj koloni matrice je onaj koji se pojavljuje najveći broj puta.
- Konsenzus niska predstavlja nisku koja se dobija nadovezivanjem najpopularnijih nukleotida iz svake kolone.
- Skor predstavlja broj nepopularnih simbola (onih koji nisu najpopularniji u svojoj koloni) u matrici.

Slika 2.9: Matrica motiva, konsenzus niska i skor



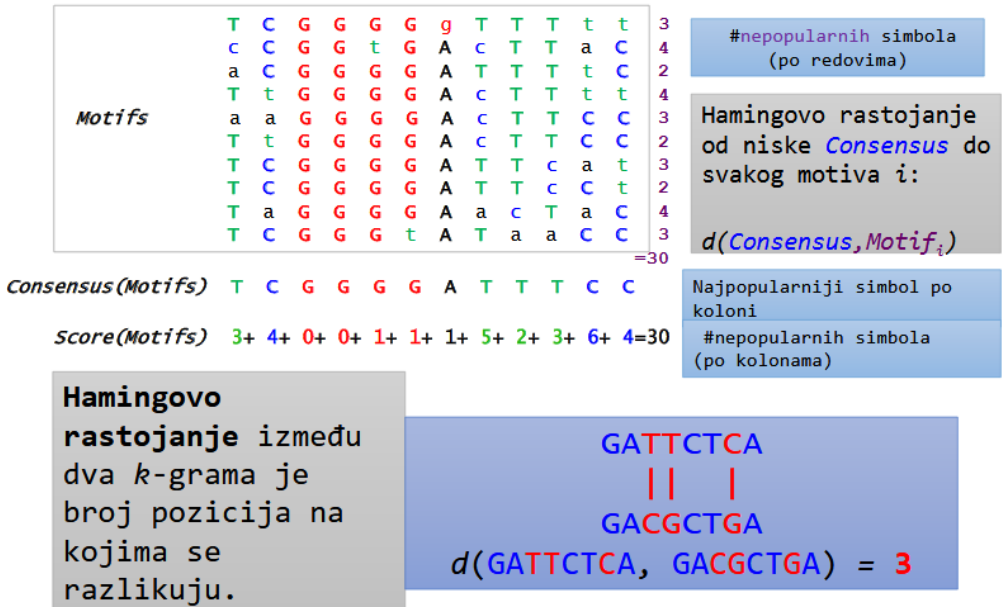
Sada ćemo preformulisati naš problem:

Problem pronalaženja motiva: Za dati skup niski iz Dna, naći skup k-grama (po jedan iz svake niske) sa minimalnim skorom među svim mogućim k-gramima iz datog skupa niski.
Ulaz: skup niski Dna i ceo broj k.
Izlaz: skup k-grama Motifs, po jedan iz svake niske Dna, tako da je vrednost skora matrice Motifs minimalna.

Prvi pokušaj bi svakako bio gruba sila. Međutim, ovaj pristup nas vodi u složenost $(n-k+1)^t$, gde je t broj niski i n dužina svake niske. Ovo je previše sporo.

Za sledeće unapređenje algoritma, primetimo najpre da je skor po vrstama i kolonama isti, tj. da je suma Hamingovih rastojanja od konsenzus niske do svakog od motiva iz matrice Motifs u stvari skor. Da bi rešili problem pronalaženja motiva, treba da nađemo način da uradimo suprotno - od konsenzus niske i niza Dna dobijemo matricu Motifs.

Slika 2.10: Matrica motiva, konsenzus niska i skor - detaljnije



Slika 2.11: Veza skora i rastojanja

$$Score(Motifs) =$$
$$\#\ \text{nepopularnih simbola po kolonama} =$$
$$\#\ \text{nepopularnih simbola po redovima} =$$
$$d(Consensus(Motifs), Motifs)$$

Predefinišimo još jednom naš problem:

Problem pronalaženja motiva - reformulacija: Naći k-gram Pattern i skup k-grama Motifs iz skupa niski Dna koji minimizuju rastojanje između svih mogućim k-grama Pattern i svih mogućih skupova k-grama Motifs.
Ulaz: skup niski iz Dna i ceo broj k.
Izlaz: k-gram Pattern i skup k-grama Motifs iz skupa niski Dna koji minimizuju d(Pattern, Motifs).

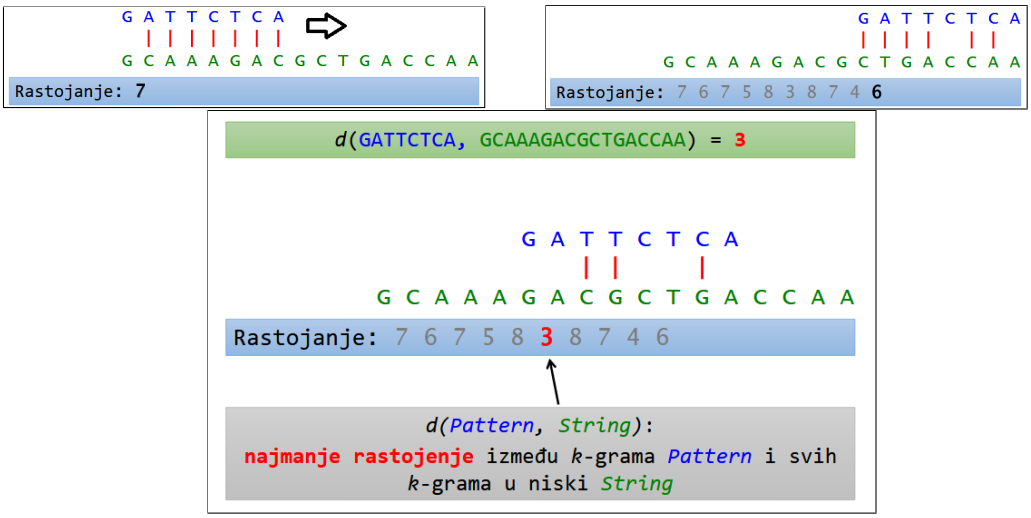
Ovo je ekvivalentno našem prethodnom problemu, jer smo primetili da d(Pattern, Motifs) predstavlja skor ukoliko je Pattern konsenzus niska. Postavlja se pitanje da li smo ovim otežali naš problem. Ispostaviće se da nismo, jer ne moramo ispitivati sve skupove k-grama Motifs, već je dovoljno da iskoristimo problem niske medijane koji ispituje sve k-grame Pattern.

2.3.4 Problem niske medijane

Pre definisanja problema, neophodno je definisati rastojanje između dve niske različite dužine, kao i još nekoliko pojmova. Hamingovo rastojanje je definisano nad dve niske istih dužina kao broj pozicija na kojima se one razlikuju. Ukoliko imamo dve niske različitih dužina, potrebno je dodefinisati Hamigovo rastojanje tako da najbolje prikaže razliku između tih niski.

- Hamingovo rastojanje između dve niske različitih dužina predstavlja minimum Hamingovih rastojanja između kraće niske i svih podniski duže niske odgovarajuće veličine.
- Definišemo rastojanje između k-grama i skupa (dužih) niski kao sumu rastojanja između tog k-grama i svih niski iz skupa.
- Niska medijana za skup niski Dna predstavlja onaj k-gram koji minimizuje rastojanje između tog k-grama i skupa Dna - d(k-gram, Dna).

Slika 2.12: Hamingovo rastojanje između dve niske različitih dužina



Sada kada smo uveli sve neophodne pojmove, možemo definisati problem.

Problem niske medijane: pronaći nisku medijanu.
Ulaz: skup niski Dna.
Izlaz: k-gram $k-mer$ koji minimizuje rastojanje d($k-mer$, Dna).

Slika 2.13: Algoritam MedianString

```
MedianString(Dna, k)
    best-k-mer ← AAA · · · AA
    for each k-mer from AAA · · · AA to TTT · · · TT
        if d(k-mer, Dna) < d (best-k-mer, Dna)
            best-k-mer ← k-mer
    return(best-k-mer)
```

Hajde da analiziramo složenost i vidimo da li smo napredovali. Ako je t dužina niza Dna, n dužina niske iz Dna, tada nam je za izračunavanje rastojanja između nekog k-grama potrebno $k*(n-k+1)$ poređenja, što nas dovodi do ukupne složenosti algoritma: $4^k*n*t*k$. Primećujemo da je ovo dobar napredak u odnosu na $n^t * k * t$, ali i dalje imamo eksponencijalnu složenost. Dakle, iako je MedianString algoritam mnogo brži od grube sile, za velike k je i dalje prespor.

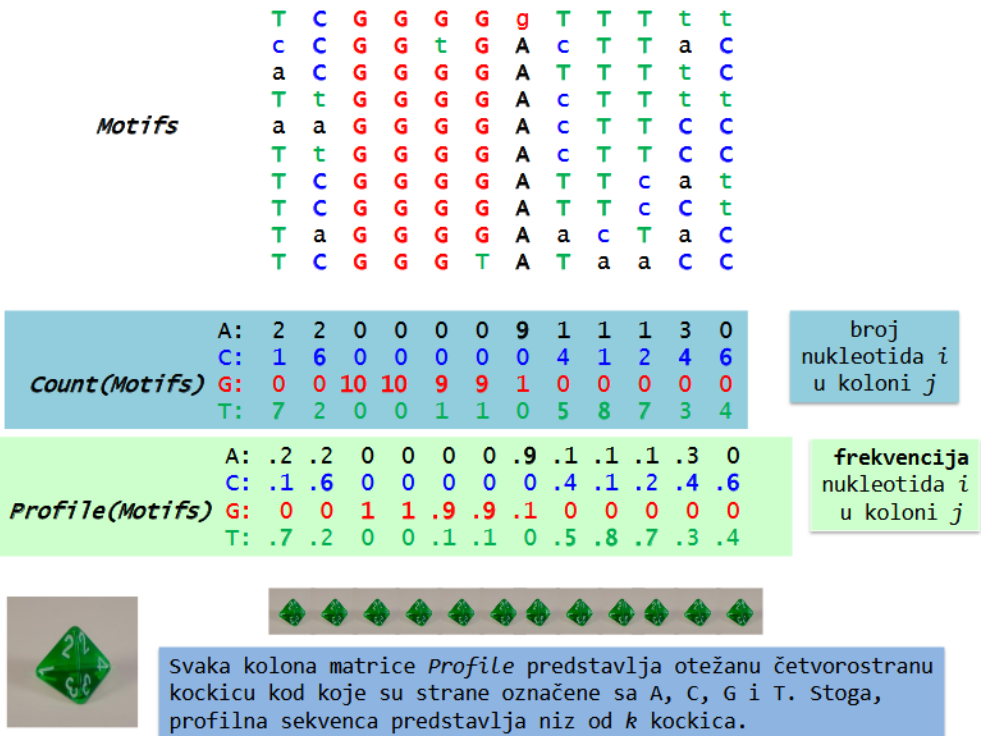
2.3.5 Probabilistički pristup

Uopšteno o pristupu

Za ovaj pristup, biće nam potrebna dva nova pojma.

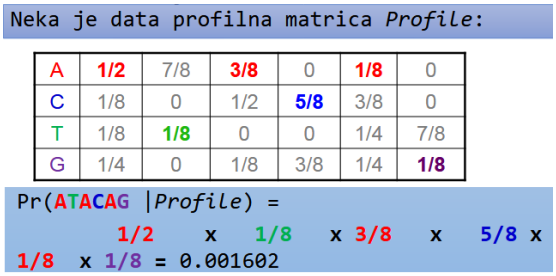
- Count matrica neke matrice motiva prikazuje koliko se puta koji nukleotid ponavlja u svakoj koloni matrice motiva.
- Profilna matrica neke matrice motiva prikazuje učestalost pojavljivanja nukleotida u svakoj koloni matrice motiva.

Slika 2.14: Count i profilna matrica



Osnovna ideja ovog pristupa može se ilustrovati bacanjem k četvorostranih kockica sa otežanim stranama - svakoj strani je pridružena verovatnoća zasnovana na koloni profilne matrice te kockice. Bacanjem ovih k kockica dobijamo nisku čiju verovatnoću da je to konsenzus niska možemo izračunati na osnovu profilne matrice na način ilustrovan na sledećoj slici:

Slika 2.15: Računanje verovatnoće k-grama



Dakle, suština pristupa leži u tome da najveća verovatnoća označava dobrog kandidata za ubačeni motiv.

Slika 2.16: Primer: najverovatniji 6-gram

6-mer	Pr (6-mer|*Profile*)	
CTATAAACCTTACAT	1/8 x 1/8 x 3/8 x 0 x 1/8 x 0	0
CTATAAACCTTACAT	1/2 x 7/8 x 0 x 0 x 1/8 x 0	0
CTATAAACCTTACAT	1/2 x 1/8 x 3/8 x 0 x 1/8 x 0	0
CTATAAACCTTACAT	1/8 x 7/8 x 3/8 x 0 x 3/8 x 0	0
CTATAAACCTTACAT	1/2 x 7/8 x 3/8 x 5/8 x 3/8 x 7/8	.0336
CTATAAACCTTACAT	1/2 x 7/8 x 1/2 x 5/8 x 1/4 x 7/8	.0299
CTATAAACCTTACAT	1/2 x 0 x 1/2 x 0 1/4 x 0	0
CTATAAACCTTACAT	1/8 x 0 x 0 x 0 x 0 x 1/8 x 0	0
CTATAAACCTTACAT	1/8 x 1/8 x 0 x 0 x 3/8 x 0	0
CTATAAACCTTACAT	1/8 x 1/8 x 3/8 x 5/8 x 1/8 x 7/8	.0004

Važna napomena: Za postizanje najbolje tačnosti probabilističke algoritme treba pokretati više puta i uzeti najbolji rezultat.

Algoritam Greedy motif search

Ideja ovog algoritma je da za svaku nisku iz Dna, koristeći profilnu matricu kreiranu za motive iz ostatka niza, odredimo najverovatniji k-gram u toj niski. Rezultat je onaj skup k-grama koji ima najbolji skor.

Slika 2.17: Algoritam GreedyMotifSearch

```
GreedyMotifSearch(Dna, k, t)
    BestMotifs ← motif matrix formed by first k-mers in each string from Dna
    for each k-mer Motif in the first string from Dna
        Motif_1 ← Motif
        for i = 2 to t
            form Profile from motifs Motif_1, ..., Motif_{i-1}
            Motif_i ← Profile-most probable k-mer in the i-th string in Dna
        Motifs ← (Motif_1, ..., Motif_t)
        if Score(Motifs) < Score(BestMotifs)
            BestMotifs ← Motifs
    return BestMotifs
```

GreedyMotifSearch je brži od Median string algoritma. Povoljan je i za veće k, ali cena toga je manja tačnost. Tačnost se može poboljšati primenom Laplasovog pravila.

Randomized motif search

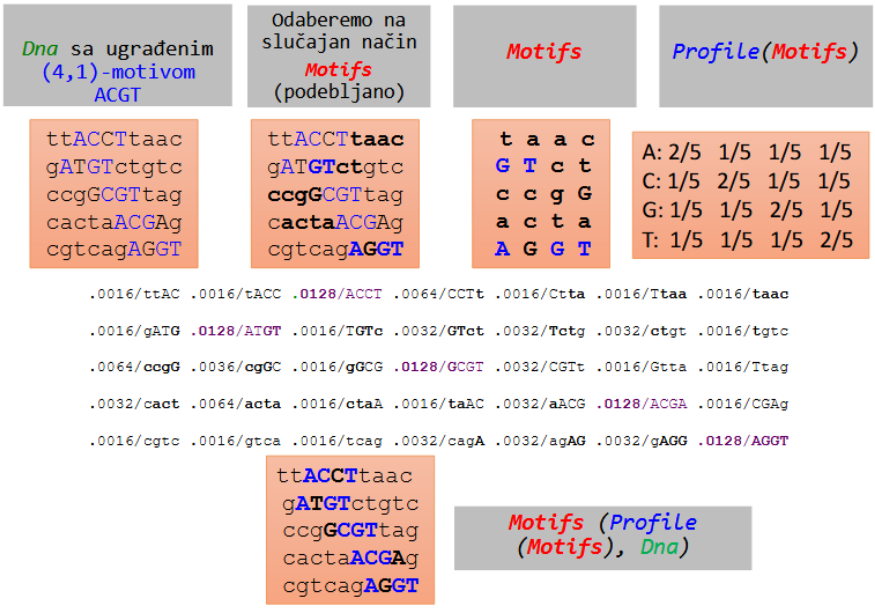
Suština ovog algoritma je da u velikom broju iteracija ažuriramo matricu motiva i profilnu matricu tako da dobijamo sve verovatniji skup motiva. Zaustavljamo se kada postignemo najniži skor. Velika prednost ovog algoritma je u nasumičnom izboru početnog skupa motiva, što nam omogućava da ga pokrećemo iznova i izaberemo najbolje rešenje.

Slika 2.18: Algoritam RandomizedMotifSearch

```
RandomizedMotifSearch(Dna, k, t)
  randomly select k-mers Motifs = (Motif_1, ... ,Motif_t) in each string from DNA
  bestMotifs ← Motifs
  while forever
    Profile ← Profile(Motifs)
    Motifs ← Motifs(Profile, Dna)
    if Score(Motifs) < Score(bestMotifs)
      bestMotifs ← Motifs
    else
      return(bestMotifs)
```

Primer rada ovog algoritma prikazan je na sledećoj slici:

Slika 2.19: Primer rada algoritma RandomizedMotifSearch



Gibsovo sempliranje

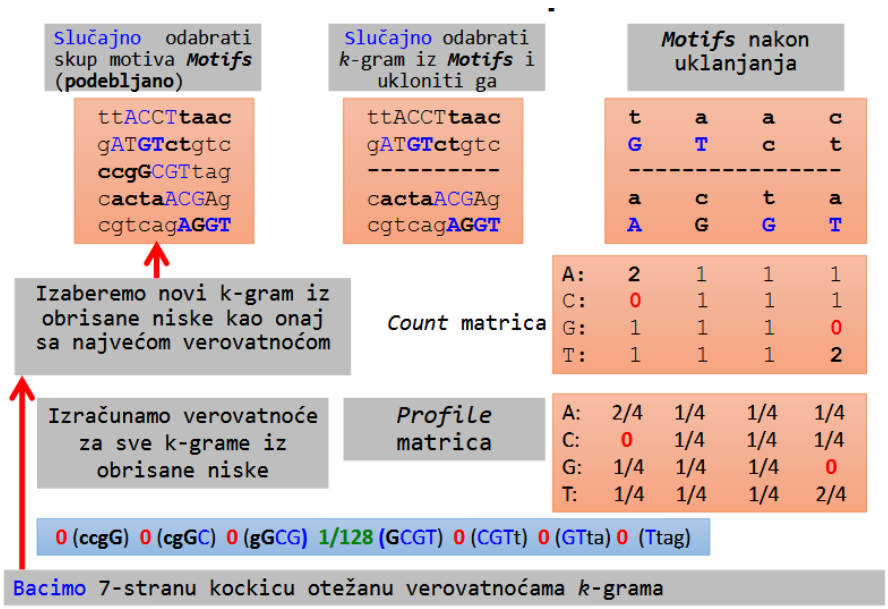
Gibsovo sempliranje predstavlja modifikaciju prethodnog algoritma u kojoj se umesto svih motiva u jednoj iteraciji ažurira samo jedan motiv, a profilna matrica kreira za sve motive izuzev tog. Možda na prvi pogled deluje kao da imamo više iteracija, ali treba napomenuti da, za razliku od prethodnog algoritma, ovde proveravamo skor posle manjih izmena, što sprečava izmenu onih motiva koji su već dobro izračunati u prethodnim iteracijama.

Slika 2.20: Algoritam Gibsovo sempliranje

1. Formirati *Motifs* izborom jednog *k*-grama iz svake sekvence na slučajan način
2. Na slučajan način odabrati jedan od *k*-grama i ukloniti ga iz *Motifs*; označimo sekvencu kojoj taj k-gram pripada sa *RemovedSequence*
3. Kreirati profilnu matricu *Profile* od preostalih *k*-grama u *Motifs*.
4. Za svaki *k*-gram iz *RemovedSequence*, izračunati *Pr(k-mer|Profile)*; na taj način dobijamo *n-k+1* verovatnoća: $p_1, p_2, \ldots, p_{n-k+1}$.
5. Bacimo kockicu sa *n-k+1* strana kod koje je verovatnoća da će pasti na i-tu stranu proporcionalna verovatnoći p_i.
6. Odredimo *k*-gram iz sekvence *RemoveSequence* kao onaj koji ima najveću verovatnoću i dodamo ga u *Motifs*.
7. **Ponavljamo korake 2-6.**

Primer rada ovog algoritma prikazan je na sledećoj slici:

Slika 2.21: Primer rada algoritma Gibsovo sempliranje



Poboljšanje Gibsovog sempliranja može se dobiti primenom Kromvelovog pravila: treba izbegavati verovatnoće 0 i 1. Ovo se može postići primenom Laplasovog pravila.

Laplasovo pravilo: U malim skupovima podataka uvek postoji šansa da se događaj koji je moguć ne desi. Slučajni algoritmi uvode pseudovrednosti koje povećavaju verovatnoće retkih događaja i eliminišu frekvencije jednake nuli zabeležene na osnovu iskustva.

Suština Laplasovog pravila leži u tome da ukoliko znamo da se događaj nekada u prošlosti desio, on ne može imati verovatnoću nula, stoga u računanje, pored vrednosti dobijenih u našem eksperimentu, ubacujemo i dve pseudovrednosti: događaj se desio i događaj se nije desio.

Slika 2.22: Računanje uslovne verovatnoće bez primene Laplasovog pravila.

Ako su $X_1, \ldots, X_{n+1}$ uslovno nezavisne slučajne logičke promenljive (neuspeh 0, uspeh 1), tada:

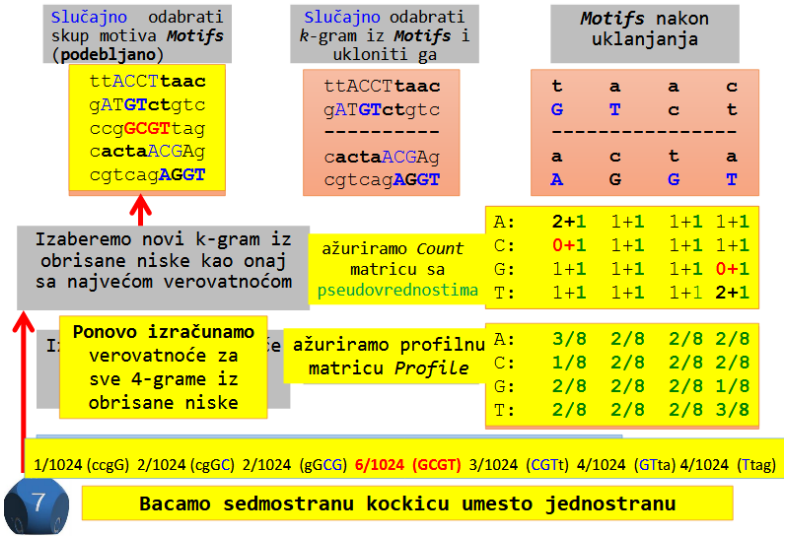
$$Pr(X_{n+1}=1 \mid X_1+\ldots+X_n=s\ )=s/n$$

Slika 2.23: Računanje uslovne verovatnoće sa primenom Laplasovog pravila.

Ako su $X_1, \ldots, X_{n+1}$ uslovno nezavisne slučajne logičke promenljive (neuspeh 0, uspeh 1), tada:

$$Pr(X_{n+1}=1 \mid X_1+\ldots+X_n=s)=(s+1)/(n+2)$$

Slika 2.24: Primer rada Gibsovog sempliranja sa primenom Laplasovog pravila



Prednost Gibsovog algoritma se ogleda u najmanjem skoru. Mana ovog algoritma je zaglavljivanje u lokalnom minimumu usled pretrage ograničenog skupa rešenja.

2.3.6 Koji princip odabrati?

Odgovor na ovo pitanje je - nema pravila. Nekada će se bolje pokazati jedan, a nekada drugi. Najbolje je da isprobamo više algoritama i vidimo koji se najbolje ponaša u našem slučaju. Možemo koristiti već zapažene prednosti i mane u izboru, kao na primer biranje Median string algoritma pri malim vrednostima k, ali isprobavanje je ipak najpouzdaniji pristup.

2.4 Zadaci sa vežbi

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

2.4.1 MedianString

```
def number_to_symbol(n):
    pairs = {
            0: 'A',
            1: 'T',
            2: 'C',
            3: 'G'
        }

    return pairs[n]


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

    prefix_index = n // 4
    r = n - n // 4
    symbol = number_to_symbol(r)

    return number_to_pattern(prefix_index, k - 1) + symbol


def hamming_distance(pattern_p, pattern):
    k = len(pattern_p)

    distance = 0

    for i in range(k):
        if pattern_p[i] != pattern[i]:
            distance += 1

    return distance


# Sumiranje hamingovih rastojanja izmednju pattern niske i svih DNK sekvenci
def d(pattern, dna):
```

```
    k = len(pattern)
    distance = 0

    for dna_string in dna:

        h_dist = float('inf')

        for i in range(len(dna_string) - k):

            pattern_p = dna_string[i:i+k]
            dist = hamming_distance(pattern_p, pattern)

            if dist < h_dist:
                h_dist = dist

        distance += h_dist
    return distance

# Pronalazenje median niske
def median_string(dna, k):
    distance = float('inf')
    median = ''

# Za svaku od 4^k niski pretpostavimo da je median niska i proverimo kolika
# je njena udaljenost od DNK sekvenci
    for i in range(4**k):
        pattern = number_to_pattern(i, k)

        current_distance = d(pattern, dna)

    # Ako je tekuca kandidat median niska bolja od dosadasnje najbolje
    # vrednosti se azuriraju i pamti se najbolja
        if distance > current_distance:
            distance = current_distance
            median = pattern

    return median


def main():
    dna = [
            'GTAGATGTCATTAGCATGCAC',
            'CCTAGCCACTCTGCCATGTCG',
            'AACTCGTGCATTCTACGACTG',
            'AAACTTTCCGGATCTTCATAC',
            'CTACATCATCGAAGGCTACGC'
        ]

    print(median_string(dna, 4))


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

2.4.2 GreedyMotifSearch

```
import copy

def symbol_to_number(n):
    pairs = {
            'A': 0,
            'T': 1,
            'C': 2,
            'G': 3
        }

    return pairs[n]

# Formiranje profil matrice za zadati skup motiva
def profile_from_motifs(motifs, k, t):
    profile = [[1 for i in range(k)] for x in range(4)]

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

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

    return profile


# Izracunavanje verovatnoce pojave pattern sekvence u odnosu na zadati profil
def probability(pattern, profile):
    prob = 1

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

        prob *= profile[index][j]

    return prob

# Pronalazenje najverovatnijeg podstringa duzine k iz zadate DNK sekvence
# koji je najverovatniji u odnosu na zadati profil
def most_probable_k_mer(dna_string, profile, k):

    best_k_mer = ''
    best_probability = -1

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

```
        if pattern_prob > best_probability:
            best_probability = pattern_prob
            best_k_mer = pattern

    return best_k_mer

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

    total_score = 0

    for j in range(k):

        counts = [0, 0, 0, 0]

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

        max_index = 0

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

        total_score += t - counts[max_index]

    return total_score


# Pohlepno pronalazenje motiv niski
def greedy_motif_search(dna, k, t):

# Pretpostavimo da najbolji skup motiv sekvenci predstavljaju
# prefiksi duzine k svih niski
    best_motifs = [dna_string[0:k] for dna_string in dna]

# i izracunamo njihov ukupan skor
    best_score = score(best_motifs, k)

    first_string = dna[0]

    motifs = []

    for i in range(len(first_string) - k):

    # Za svaku podnisku duzine k iz prve DNK sekvence kazemo da u tekucoj
    # iteraciji predstavlja prvi motiv

        motifs.append(first_string[i:i+k])
```

```
    # Iz svake od preostalih t-1 DNK sekvenci izdvajamo podstring duzine k
    # koji je najverovatniji u odnosu na profil dobijen od motiva dobijenih iz
    # prethodnih iteracija. Taj podstring dodajemo u skup motiva kako bi se
    # koristio u narednoj iteraciju za pronalazenje sledeceg motiva

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

    # Sa svaki izgenerisami skup motiva proveravamo da li daje bolji skor
    # u odnosu na do sada najbolji pronadjen
        current_score = score(motifs, k)

    # Ako je trenutni skup motiva bolji od dosadasnjeg azuriraju se vrednosti
        if current_score < best_score:
            best_motifs = copy.deepcopy(motifs)
            best_score = current_score

    return best_motifs

def main():
    dna = [
            'GTAGATGTCATTAGCATGCAC',
            'CCTAGCCACTCTGCCATGTCG',
            'AACTCGTGCATTCTACGACTG',
            'AAACTTTCCGGATCTTCATAC',
            'CTACATCATCGAAGGCTACGC'
        ]

    print(greedy_motif_search(dna, 4, len(dna)))


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

2.4.3 RandomizedMotifSearch

```
import copy
import random

def symbol_to_number(n):
    pairs = {
            'A': 0,
            'T': 1,
            'C': 2,
            'G': 3
        }

    return pairs[n]

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

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

        prob *= profile[index][j]

    return prob

# Izdvajanje pseudoslucajno odabranih podniski iz DNK sekvenci u skupu
def random_k_mers(dna, k, t):
    k_mers = []

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

    return k_mers


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

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

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

    return profile

# Pronalazenje podniski DNK sekvenci iz skupa koje su najverovatnije u
# odnosu na zadati profil i one zajedno cine skup motiva
def motifs_from_profile(profile, dna):
    motifs = []
    k = len(profile[0])

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

    return motifs


def score(motifs, k):
    t = len(motifs)

    total_score = 0

    for j in range(k):
```

```
        counts = [0, 0, 0, 0]

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

        max_index = 0

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

        total_score += t - counts[max_index]

    return total_score

def most_probable_k_mer(dna_string, profile, k):

    best_k_mer = ''
    best_probability = -1

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

        if pattern_prob > best_probability:
            best_probability = pattern_prob
            best_k_mer = pattern

    return best_k_mer

# Pronalazenje motiva koriscenjem algoritma za pseudoslucajni izbor
def randomized_motif_search(dna, k, t):

# Pretpostavimo da najbolji skup motiva cine slucajno odabrane podniske
# iz skupa DNK sekvenci
    motifs = random_k_mers(dna, k, t)
    best_motifs = copy.deepcopy(motifs)
    best_score = score(best_motifs, k)

# Dok se skor popravlja svakom iteracijom:
    while True:

    # Formiramo profil od tekucih motiva
        profile = profile_from_motifs(motifs, k, t)

    # A zatim motive od dobijenog profila
        motifs = motifs_from_profile(profile, dna)

        current_score = score(motifs, k)

        if current_score < best_score:
```

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


def main():
    dna = [
            'GTAGATGTCATTAGCATGCAC',
            'CCTAGCCACTCTGCCATGTCG',
            'AACTCGTGCATTCTACGACTG',
            'AAACTTTCCGGATCTTCATAC',
            'CTACATCATCGAAGGCTACGC'
        ]

    print(randomized_motif_search(dna, 4, len(dna)))


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

2.4.4 GibbsSampler

```
import copy
import random

def symbol_to_number(n):
    pairs = {
            'A': 0,
            'T': 1,
            'C': 2,
            'G': 3
        }

    return pairs[n]

def probability(pattern, profile):
    prob = 1

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

        prob *= profile[index][j]

    return prob

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

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

```
        k_mers.append(dna_string[start:start+k])

    return k_mers

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

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

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

    return profile

def score(motifs, k):
    t = len(motifs)

    total_score = 0

    for j in range(k):

        counts = [0, 0, 0, 0]

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

        max_index = 0

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

        total_score += t - counts[max_index]

    return total_score

def most_probable_k_mer(dna_string, profile, k):

    best_k_mer = ''
    best_probability = -1

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

        if pattern_prob > best_probability:
            best_probability = pattern_prob
```

```
            best_k_mer = pattern

    return best_k_mer

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

    for j in range(N):

        # Biramo slucajno i iz skupa [0,t)
        i = random.randrange(0,t)

        # Formiramo skup motiva koji se sastoji od svih dosadasnjih motiva osim
    ↪  i-tog
        selected_motifs = copy.deepcopy(motifs)
        del selected_motifs[i]

        # Pravimo profil od odabranih motiva (bez i-tog)
        profile = profile_from_motifs(selected_motifs, k, t-1)

        # Za i-ti motiv postavljamo najverovatniji podstring duzine k iz i-te
    ↪ DNK sekvence, u odnosu na dobijeni profil
        motifs[i] = most_probable_k_mer(dna[i], profile, k)
        del selected_motifs

        # Ako dobijeni motiv ima skor bolji od do sada najboljeg, vrednosti se
    ↪ azuriraju
        current_score = score(motifs, k)

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

    return best_motifs

def main():
    dna = [
            'GTAGATGTCATTAGCATGCAC',
            'CCTAGCCACTCTGCCATGTCG',
            'AACTCGTGCATTCTACGACTG',
            'AAACTTTCCGGATCTTCATAC',
            'CTACATCATCGAAGGCTACGC'
        ]

    print(gibbs_sampler(dna, 4, len(dna), 500))


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

Literatura