DNA patterns that play the role of Molecular Clocks. (Randomized algos).

- Daily schedules of animals, plants, and even bacteria are controlled by "time keeper" called circadian clock.

- How do individual cells in animals and plants let alone in bacteria know when they should slow down or "increase" the production of certain proteins. is there a "clock gene"

- genes responsible for "breaking" the circadian clock to cause DSPS.

- Today many circadian genes have been discovered these genes having names like timeless, clock, cycle "orchestrate" the behavior of other genes and display high degree of evolutionary conservation across speci.

- Every plant cell keeps track of day and night separately / independent of other cells, and that just three plants genes, called LHY, CCA1 and TOC1 are Clock's master time keepers.
Regulatory genes and the regulatory proteins that they "encode", are often controlled by external factors, in order to allow organisms to adjust their gene expression.

- TOC1 promotes the expression of LHY and CCA1.
- LHY, CCA1 and TOC1 are able to control the "transcription" of "other genes" because the "regulatory proteins" that they encode are "transcription factors" or "master regulatory proteins" that turn other genes ON/OFF.

o) Master Regulatory proteins that turn other Genes ON/OFF. (Also, called Transcription Factors)

o) A transcription factor regulates a gene by binding to a specific short DNA interval called a "regulatory motif" or "transcription

A

C
G
T

factor binding site" in the gene's upstream region (600 – 1000 bp region preceding the start of gene).

example: CCA1 binds to AAAATCT in upstream regions of many genes regulated by CCA1.

Regulatory Motif

=> How to locate regulatory Motifs without knowing how they look:

they may vary though.

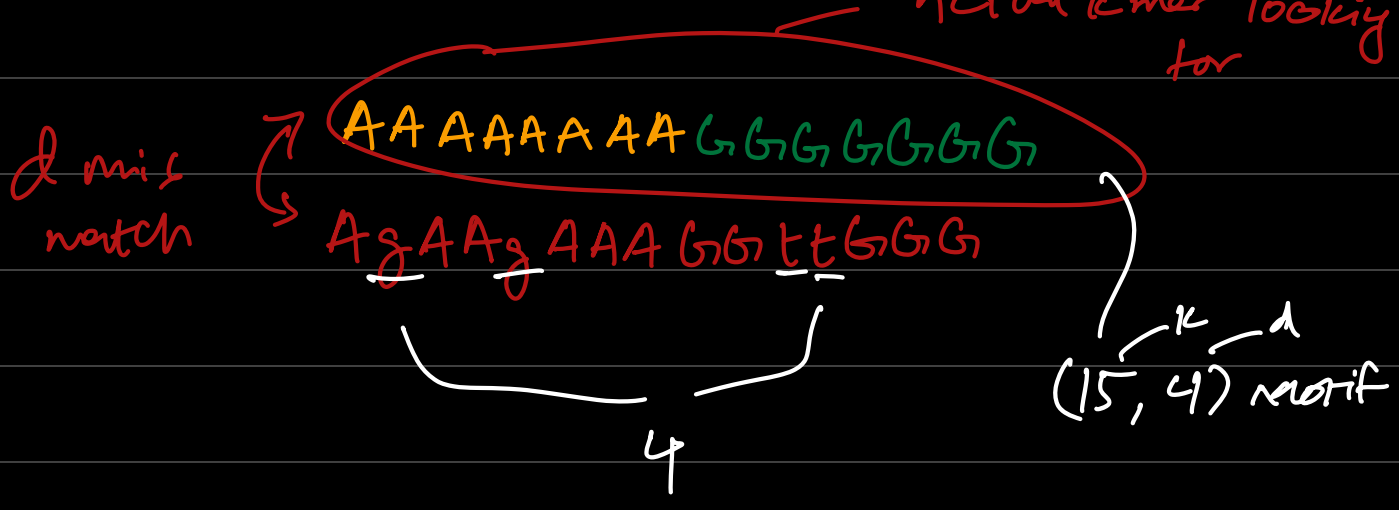
=> Motif finding: discovering the hidden message "shared" by collection of strings. called "Regulatory Motifs" or binding site for transcription factors which are "master regulatory proteins". (can turn other genes ON / OFF).

⇒ Regulatory Motif is a "pattern" that "appears atleast once" (perhaps with variations or Mutations) in "each" of the many different "regions" that are scattered throughout the genome.

↳ strings

exact d mismatches rather <= may be

==



Generate all k-mers with d mismatch.

Any (k, d) – motif must be at most d mismatches apart from some kmer appearing in the first string of DNA.

```
def Immediate_Neighbors(pattern):
    neighborhood = set()
    for i in range(0, len(patter)):
        symbol = pattern[i] # ith Nucleotid.
        for c in "AGTC": # for each nucleotid.
            if c != symbol:        # x diffrent from symbol
                .add(pattern.replace(symbol, c))

    return neighborhood


def iterative_neighbors(pattern, d):
    neighborhood = set()
    neighborhood.add(pattern)
    for i in range(0, d):
        neighbors_copy = neighborhood.copy()
        for pat in neighbors_copy:
            neighbors = Immediate_Neighbors(pat)
            neighborhood.update(neighbors)
```

Sort of like union

or

```
            neighborhood != neighbors
```

inplace union.

return neighborhood

```
def Neighbors(pattern, d):
    if d = 0
        return {pattern}
    if len(pattern) == 1:
        return {A, C, G, T}
    neighborhood = set()
    suffix = Neighbors(pattern[1:], d)
    for text in suffix:
        if hammingdistance(pattern[1:], text) < d:
            for c in "AGTC":
                new = c + text
                neighborhood.add(new)
        else:
            new = pattern[0] + text
```

A|CT
C|T

```
        neighborhood.add(new)

    return neighborhood.
```

distance b/w all patterns and strings

```
def DistanceBetweenPatternANDSTRINGS(pattern, dna):

    K = len(pattern)
    distance = 0
    for gene in dna:          # gene string
        hamming_distance = float('inf')
        for i in range(0, len(gene) - K + 1):
            if hamming_distance > Hamming_Distance
                (pattern, gene[i:i+K])
                hamming_distance = 2
        distance = Distance + hamm_distan.
```

return distance

```
def median_string(dna, k):
    distance = float('inf'), median = ""
    all_kmers = generate_kmers(k, ['A','T','G','C'])
    for kmer in all_kmers:
        if distance > d(kmer, dna)
            distance = d(kmer, dna)
            Median = kmer

return median
```

A

C

G

T

$Pr(ACGGGGATTACC \mid Profile)$

↓

probability from profile matrix

$0.2 \times 0.6 \times 1 \times \ldots = 0.00083\ldots$

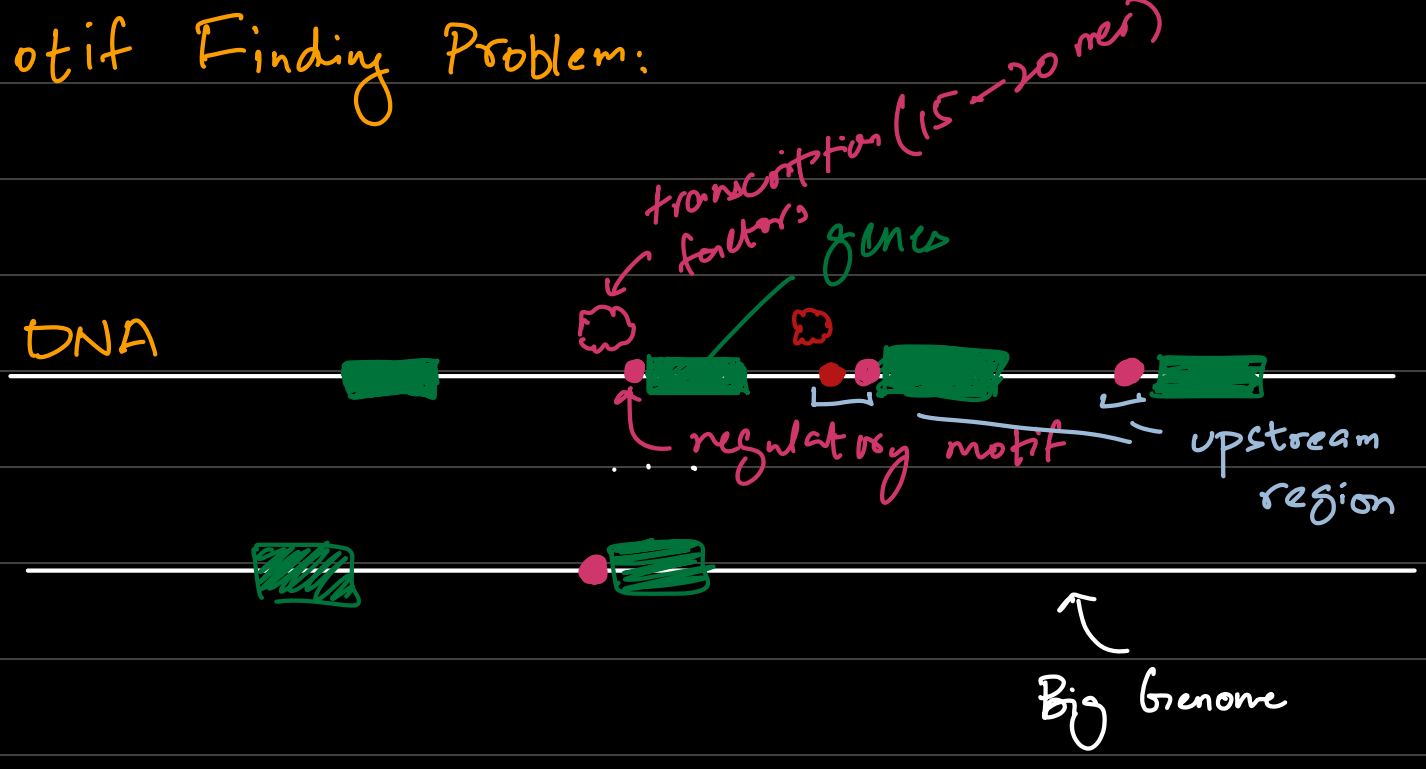
Look for Kmers that have larger values of Pr (text | Profile).

```
def profile_most_probable_kmer(text, k, profile_matrix)

    mapping = {'A': 0, 'C': 1, 'G': 2, 'T': 3}
    k_mer_probability = { }

    for i in range(len(text) - k + 1):
        k_mer = text[i : i+k], probability = 1
        for j in range (len(k_mer)): 0 -> 4
            probability *= profile_matrix[k_mer[j]][j]
```

k_mer_probability[k_mer] = probability.

maximum = max(k_mer_probability.values())
return key with this maximum.

Motif Finding Problem:



- We might know that some collection of genes are related to function e.g. immunity genes. When organism sicle all the immunity genes "turn on" at the same time and "start producing proteins".

- How do all those genes get regulated & work together.

- There are short snippets (15 – 20 bp long) of sequence before the gene that can be bound by the same transcription factor.

- When transcription factor binds to a regulatory motif, the "expression" of the gene changes (can go up or down).

- A transcription factor is conductor giving signal to bunch of different genes and regulating them.

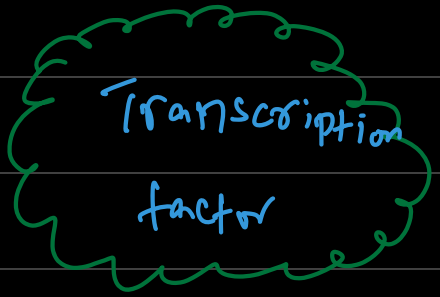
- A same gene can be regulated by multiple transcription factors (e.g. may be a gene is related to "recovering from sickness" as well as "pregnancy").

- If we know that several genes are related to same event (e.g their expression

changes after athletic activity) we can "search" for "common motifs" in "upstream" of these genes.

- If we find correct motif it may enable us to identify additional genes regulated by the "transcription factors."

- Transcription factors don't always bind perfectly to regulatory motifs.

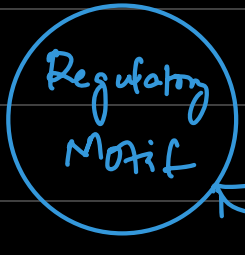


Some Mismatches are allowed.

ATTTACCGTGC

Perfectly match → TAAATGGCACG

ideal situation as reverse complements each other.



have to find them.

- 10 sequences / strings of DNA
- Where 15 - mer inserted in each sequence at random location with 4 random mismatches.
- The true motif is "AAAAAAAAAGGGGGGGGG" (Consensus Motif)
  but exact sequence never appears.

strings or upstream strings → generate k-mers of some length (5, 1)-motif → that kmers nearest
(k, d)

neighbors are generated with d mismatch ⇒ one we call such "k mers".

we iterate through every kmer and check if it appears once with d mismatch in every sequence or string of DNA. Then it is (k, d) motif, but this algo is slow.

- Collection of Motifs ⇒ Give you a profile matrix

- Using profile matrix we can calculate the probability of every kmer or motif,

- The profile matrix gives the probability of generating the kmer or motif.

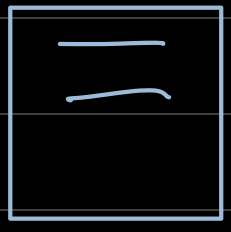
- Kmer has higher probability when it is similar to consensus string of a profile.

◦) Profile - Most Probable kmer.

⇒ Greedy Motif Search:

Starts by forming a motif matrix from a arbitrary "selected kmers" in each string of DNA. (in our implementation best motifs are first kmer in each string). The algorithm then attempts to

improve the initial motif matrix by trying each of the kmers in Dna_1 as the first motif. For given kmer $Motif_1$ in Dna_1, it builds a profile matrix "Profile" for this Initial Motifs.



one kmer, and sets $Motif_2$ equal to the profile-most probable kmer in Dna_2. updating profile as the profile matrix found for motifs

$$motif_3 = \text{Profile most probable}\left(\begin{bmatrix} motif_1 \\ motif_2 \end{bmatrix}\right)$$

After obtaining kmer from each String to obtain collection Motifs. — algo tests to see whether Motifs outscore the current best scoring collection of motifs and then moves $Motif_1$ one symbol one in Dna_1 begining the entire process of generating motifs again.

RandomizedMotifSearch(Dna, k, t):

Motifs ← randomly chosen kmers from each string in Dna

BestMotifs ← Motifs

While True:

Profile ← Profile(Motifs) — Profile matrix of the Motifs

Motifs ← profile most probable(profile, Dna) kmer

append

if Score(Motifs) < Score(BestMotifs):

BestMotifs ← Motifs

else:

return BestMotifs

Rule of Succession:

Just because we have not observed an event does not mean that we should assign its future probability to zero.

Add pseudocount value to the counts of each type of event before normalizing.

Motifs: T A A C

G T C T

A C T A

A G G T

Profile:

A	2/4	1/4	1/4	1/4
C	0	1/4	1/4	1/4
G	1/4	1/4	1/4	1/4
T	1/4	1/4	1/4	2/4

Adding a pseudocount of 1 produces following count and profile matrix.

— original

Count (matrix)

A: 2+1 1+1 1+1 1+1

C: 0+1 1+1 . . . . .

G: . . . . . .

T: . . . . .

Profile Matrix

3/8 2/8 2/8 2/8

. . . . .

. . . . .

. . . . .

Modify Create Profile Matrix to account for pseudocounts.

↳ add one to each entry of counts.

$$\text{profile} = \text{count}/(t+4)$$

"↓ 3 - row TTACC"

seq1 = TTACCTTAAC

motifs = [TTAC]

profile =

A	0.2 1/5	0.2 1/5	0.4 1+1/5	0.2 1/5
C	0.2 1/5	0.2 1/5	0.2 1/5	0.4 1+1/5
G	0.2 1/5	0.2 1/5	0.2 1/5	0.2 1/5
T	0.4 1+1/5	0.4 1+1/5	0.2 1/5	0.2 1/5

*) After we have found collection of motifs ⇒ we can find consensus of those motifs for true motif.

*) Looking for hidden messages:

If collection of genes are implicated in the same function (e.g circadian clock or responsible), then a single transcription factor may "bind" to the same "keyword" in many of the genes' upstream regions, perhaps with "minor variations".

↳ regulatory motifs
↳ they will have slight variations.

⇒ Randomized Algorithms:

o) Randomized Algorithms use randomness to achieve their desired outcomes.

Las Vegas Algorithms: results are guaranteed to be correct, but the amount of time the algo will take is random

Ex: Quick Sort Algos

Monte Carlo Algorithms: rely on repeated sampling of results. NOT guarenteed to be correct. (fast).

Ex: Finding the probability of biased die landing on "each side" by rolling it many times.

↳ estimating probability of landing on each side.

def RandomizedMotifSearch(Dna, k, t):

Motifs ← randomly select k-mers motifs from each sequence

bestMotifs ← Motifs

While (forever):

Profile ← Profile(Motifs)

Motifs ← Motifs(profile, Dna)

↳ Profile most probable k-mers

it evaluate through each sequence one by one.

new set of

Motifs of profile Mat probable kmers.

if score(Motifs) < score(Best Motifs)

Best Motifs ← Motifs

else:

return Best Motifs

- Run 1000
- Use Pseudo Counts

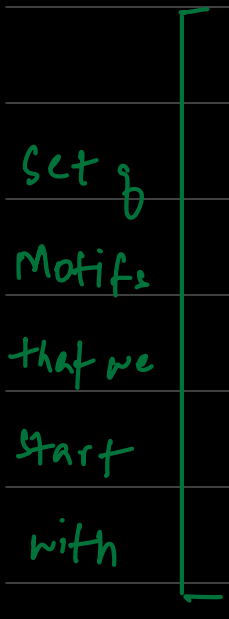
(*) If initial random set of motifs includes at least one of the 'correct' motifs, the profile matrix (slightly) biased towards the correct motif

- This bias points the search in the right direction.

⇒ Gibbs Sampling:

- This is a Markov-Chain Monte Carlo (MCMC) method.
- is method to "draw samples" from a "complex probability distribution" by breaking it down into "simpler pieces"
  - ◦) Have Unknown things to estimate (variables $x_1, x_2, x_3 \ldots$)
  - ◦) Instead trying to figure out all at once, fix all but one variable, and draw sample for that single variable based on the others.

- Rotate through each variable, each time sampling the new value from its "Conditional distribution" given the latest values of all the other variables.
- By cycling the variables many times, the collected samples start to reflect the true underlying (joint) distribution you're interested in.



→ One k-mer changed.

* Gibbs Sampling will "change" only a "single k-mer" in each step, as well as changing this k-mer more liberally.

DNA Sequences

$$\begin{bmatrix} ttACCTtaac \\ gATGTctgtc \\ ccgGtCGTtag \end{bmatrix} \Rightarrow \begin{bmatrix} ttACCTtaac \\ \cdots\cdots \\ ccgGtCGTtag \end{bmatrix}$$

- pick random motifs of some length k from each sequence. Gibbs Sampling randomly selects one of the strings to be replaced.

- Generate Profile Matrix for the remaining motifs. Use that profile matrix and the sequence to get probability distribution for each kmer for that sequence and select one randomly weighted by the probabilities after "normalizing" them so that they sum to 1.

$$\left(\frac{1}{8}\right) \cdot \left(\frac{2}{8}\right) \cdot \left(\frac{2}{8}\right) \cdot \left(\frac{1}{8}\right) = \frac{4}{8^4}$$

- Set of probabilities for each kmer.

- Randomly sample or choose k mer randomly but it is proportional to probability given here sort of like roulette wheel. High probability more likely to get selected.

- N times.

N = 1000

GibbsSampler(*Dna*, *k*, *t*, *N*)

```
randomly select k-mers Motifs = (Motif_1, ..., Motif_t) from Dna
BestMotifs ← Motifs
for j ← 1 to N
    i ← randomly generated integer between 1 and t
    Profile ← profile formed from all Motifs other than Motif_i
    Motif_i ← Profile-randomly generated k-mer in Dna_i
    if Score(Motifs) < Score(BestMotifs)
        BestMotifs ← Motifs
return BestMotifs
```

run this whole thing 20 times

HW 2:

a) Kmers of length 15 – 20

b) More iterations.

– Markov Chain Monte Carlo

*) Gibbs Sampling:

Sample from a joint distribution that is hard to directly sample from.

- Get values for bunch of variables that are related to each other in some way.

- HealthMetrics: height, age, weight, blood pressure, etc
  ↳ generate a collection of these variable values that would be realistic for real human, without being same as data we have access to.

Some sample

Gibbs Sampling: want to generate values for $x_1 \ldots x_n$

(1) Pick initial guesses for each variable.

(2) update the value of x_i by sampling

set of random values for all metrics

the probability $\mathbb{P}(x_1 \mid x_2, x_3 \ldots x_n)$

Find the height that's most probable given the other measurement values.

(3) Update the value of X_2 by sampling $\mathbb{P}(x_2 \mid x_1, x_3 \ldots x_n)$

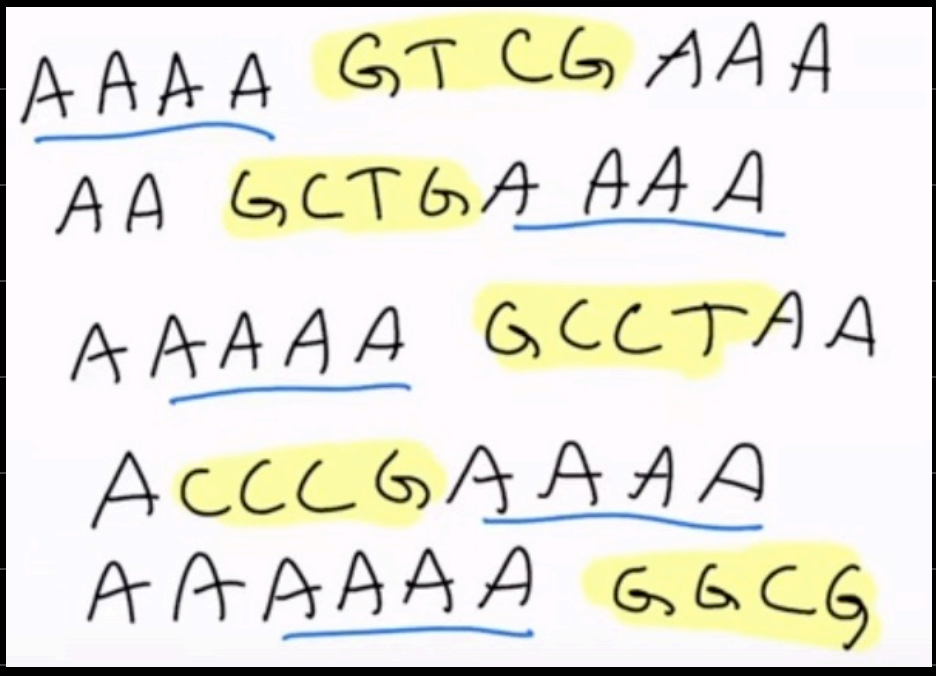
Find the most likely age for this person.

(4) Do this for all variables.

(5) Iterate many times ✓

*) wait for samples to "converge"
→ for humans dataset.

Problems: skewed may be our sequence is all As or background all A's.



⇒ scoring function: "relative entropy"

⇒ Let W = A or T WCGW

⇒ Let's us observe protein binding to DNA sequence.

o) Try identifying a set of DNA sequences with similar function, and then find motifs.

o) Try analyzing chip-seq data.

c) Try different scoring functions (entropy, relative entropy).

Sequences — strings of DNA

best motifs = all first kmers of all sequences of DNA

Start with first kmer of seq 1 (we iterate through all)

```
motifs = [kmer]
profile = profile(kmer) #Create profile
iterate through every sequence seq2 ... seqt:
    motif = profile_most_probable(profile, seq)
    add this new motif to motifs
    profile = profile(motifs) #Create profile
if score(Motifs) < score(Bestmotifs):
    Bestmotifs = Motifs
```

return best motifs.

← Like CP choosing local optimum at each step to reach the global optimum

*) Basically we start with First 1cmer of seq1 generate its profile and move to second sequence — we send that sequence and this profile to profile_most_probable_1cmer, which gives probability to every kmer of second sequence and we return one with highest probability of being generated with that profile. We add new motif or returned one to list of motifs, and create new profile of these two motifs and move to seq 3 then ⟹ same process again until we reach end of all sequences ⟹ then score these new collection of motifs to the best motifs if score is less than new motifs become best motifs.

*) This process ends when we have gone through all the kmers of seq1.

```
def score(motifs):
    consensus = ""
    rows = len(motifs)    # rows 3
    cols = len(motifs[0])  # 4
    for j in range(0, cols):
        freq = {}
        for i in range(0, rows):    motifs[j][i]
            freq[motifs[i][j]] = freq.get(motifs[i][j],0)
                                                    +1
        Consensus += max(freq, key = freq.get)

    score = 0
    for motif in motifs:
        score += hamming_distance(motif, Consensus)

    return score
```