Combinatorics and its applications to Biology

Benjamin Tetreault, Grace Holden, Ashley Burke, Ethan Miller

What is combinatorics?

Combinatorics is essentially the mathematical study of counting. However, it has many applications in communication networks. This project focuses more on its applications in genetics. For example, counting the number of permutations possible for a given gene ordering.

Enumerating Gene Order

Given: A positive Integer n <= 7

Return: The total number of permutations of length n, followed by a list of such permutations.

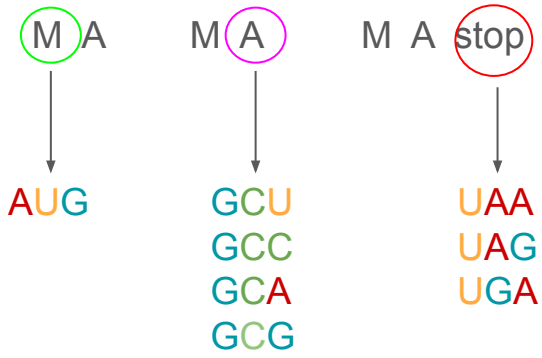
For this project we are trying to simulate how we would find all the possible ways we can rearrange a genome.

Inferring mRNA from Protein

Given: A protein string of length at most 1000 amino acids

Return: The total number of different RNA strings from which the protein could have been translated, module 1,000,000 (Don’t neglect the importance of the stop codon in protein translation)

Given Protein String: MA



Possible Combinations:

AUG GCU UAA
AUG GCU UAG
AUG GCU UGA
AUG GCC UAA
AUG GCC UAG
AUG GCC UGA

AUG GCA UAA
AUG GCA UAG
AUG GCA UGA
AUG GCG UAA
AUG GCG UAG
AUG GCG UGA

Return (Answer): 12

2nd letter in the codon

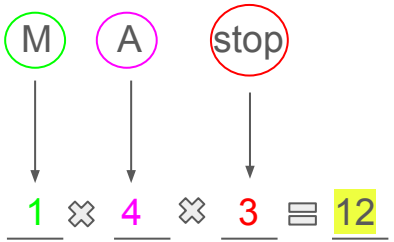
1st letter in the codon	U	C	A	G	3rd letter in the codon
U	UUU Phe (F) UUC UUA Leu (L) UUG	UCU UCC UCA Ser (S) UCG	UAU Tyr (Y) UAC UAA STOP UAG STOP	UGU Cys (C) UGC UGA STOP UGG Trp (W)	U C A G
C	CUU CUC CUA Leu (L) CUG	CCU CCC CCA Pro (P) CCG	CAU His (H) CAC CAA Gln (Q) CAG	CGU CGC CGA Arg (R) CGG	U C A G
A	AUU AUC Ile (I) AUA AUG Met (M) START	ACU ACC ACA Thr (T) ACG	AAU Asn (N) AAC AAA Lys (K) AAG	AGU Ser (S) AGC AGA Arg (R) AGG	U C A G
G	GUU GUC GUA Val (V) GUG	GCU GCC GCA Ala (A) GCG	GAU Asp (D) GAC GAA Glu (E) GAG	GGU GGC GGA Gly (G) GGG	U C A G

https://ileriseviye.files.wordpress.com/2013/12/geneticcode.jpg

Protein Amino Acid	# of codons
M (start)	1
W	2
Y	2
C	2
E	2
K	2
Q	2
S	6
L	6
R	6
G	4

Protein Amino Acid	# of codons
F	2
D	2
H	2
N	2
M	1
A	4
P	4
T	4
V	4
I	3
Stop	3

Given Protein String: MA



Downloaded Rosalind File:

```
MSTWGERGLMPYRGLACEGHI
```

Type command into terminal:

```
python ./Inferring\ mRNA\ from\ Protein.py
```

Get program answer:

```
771392
```

Open Reading Frames

Given: A DNA string *s* of length at most 1kbp in FASTA format

Return: Every distinct candidate protein string that can be translated from ORFs of *s*. Strings can be returned in any order.

Given DNA String:

AGCCATGTAGCTAACTCAGGTTACATGGGGATGACCCCGCGACTTGGATTAGAGTCTCTTTTGGAATAAGCCTGAATGATCCGAGTAGCATCTCAG

1. Transcribe DNA into RNA: *DNA*.replace('T', 'U')

AGCCA**T**G**T**AGC**T**AAC**T**CAGG**TT**ACA**T**GGGGA**T**GACCCCGCGAC**TT**GGA**TT**AGAG**T**C**T**C**TTTT**GGAA**T**AAGCC**T**GAA**T**GA**T**CCGAG**T**AGCA**T**C**T**CAG
AGCCA**U**G**U**AGC**U**AAC**U**CAGG**UU**ACA**U**GGGGA**U**GACCCCGCGAC**UU**GGA**UU**AGAG**U**C**U**C**UUUU**GGAA**U**AAGCC**U**GAA**U**GA**U**CCGAG**U**AGCA**U**C**U**CAG

2. Count # of 'AUG' substrings: *RNA*.count('AUG')

AGCC**AUG**UAGCUAACUCAGGUUAC **AUG**GGG**AUG**ACCCCGCGACUUGGAUUAGAGUCUCUUUUGGAAUAAGCCUGA **AUG**AUCCGAGUAGCAUCUCAG

3.1.1 Create Substring of RNA starting at start codon

~~AGCC~~**AUG**UAGCUAACUCAGGUUAC **AUG**GGG**AUG**ACCCCGCGACUUGGAUUAGAGUCUCUUUUGGAAUAAGCCUGA **AUG**AUCCGAGUAGCAUCUCAG

AUGUAGCUAACUCAGGUUAC **AUG**GGG**AUG**ACCCCGCGACUUGGAUUAGAGUCUCUUUUGGAAUAAGCCUGA **AUG**AUCCGAGUAGCAUCUCAG

3.2.1 Split the RNA string into codons :

[**AUG** UAG CUA ACU CAG GUU ACA UGG GGA UGA CCC CGC GAC UUG GAU UAG AGU CUC UUU UGG AAU AAG CCU GAA UGA UCC GAG UAG CAU CUC AG]

3.3.1 Cut the list at the first stop codon:

~~[**AUG** **UAG** CUA ACU CAG GUU ACA UGG GGA **UGA** CCC CGC GAC UUG GAU UAG AGU CUC UUU UGG AAU AAG CCU GAA UGA UCC GAG UAG CAU CUC AG]~~

[**AUG**]

Locations of stop codons in list: 1, 9 → min = 1

3.4.1 Add the list to the list of reading frames:

List of all reading frames = [AUG]

3.1.2 Create Substring of RNA starting at start codon:

~~AUG~~~~UAGCUAACUCAGGUUAC~~ AUGGGGAUGACCCCGCGACUUGGAUUAGAGUCUCUUUUGGAAUAAGCCUGA AUGAUCCGAGUAGCAUCUCAG

AUGGGGAUGACCCCGCGACUUGGAUUAGAGUCUCUUUUGGAAUAAGCCUGA AUGAUCCGAGUAGCAUCUCAG

3.2.2 Split the RNA string into codons:

[AUG GGG AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA UAA GCC UGA AUG AUC CGA GUA GCA UCU CAG]

3.3.2 Cut the list at the first stop codon:

[AUG GGG AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA UAA GCC UGA AUG AUC CGA GUA GCA UCU CAG]

[AUG GGG AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]

Locations of stop codons in list: 14, 16 → min = 14

3.4.2 Add the list to the list of reading frames:

List of all reading frames =

[AUG]

[AUG GGG AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]

3.1.3 Create Substring of RNA starting at start codon:

~~AUGGGG~~AUGACCCCGCGACUUGGAUUAGAGUCUCUUUUGGAAUAAGCCUGA AUGAUCCGAGUAGCAUCUCAG

AUGACCCCGCGACUUGGAUUAGAGUCUCUUUUGGAAUAAGCCUGA AUGAUCCGAGUAGCAUCUCAG

3.2.3 Split the RNA string into codons:

[AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA UAA GCC UGA AUG AUC CGA GUA GCA UCU CAG]

3.3.3 Cut the list at the first stop codon:

[AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA UAA GCC UGA AUG AUC CGA GUA GCA UCU CAG]

[AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]

Locations of stop codons in list: 12, 14 → min = 12

3.4.3 Add the list to the list of reading frames:

List of all reading frames =

[AUG]

[AUG GGG AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]

[AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]

3.1.4 Create Substring of RNA starting at start codon:

~~AUG~~ACCCCGCGACUUGGAUUAGAGUCUCUUUUGGAAUAAGCCUGA ~~AUG~~AUCCGAGUAGCAUCUCAG

AUGAUCCGAGUAGCAUCUCAG

3.2.4 Split the RNA string into codons:

[AUG AUC CGA GUA GCA UCU CAG]

3.3.4 Cut the list at the first stop codon:

[AUG AUC CGA GUA GCA UCU CAG]

[]

Locations of stop codons in list: __

3.4.4 Add the list to the list of reading frames:

List of all reading frames =

[AUG]

[AUG GGG AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]

[AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]

[]

Where are we?

AGCCATGTAGCTAACTCAGGTTACATGGGGATGACCCCGCGACTTGGATTAGAGTCTCTTTTGGAATAAGCCTGAATGATCCGAGTAGCATCTCAG

AGCCA**U**G**U**AGC**U**AAC**U**CAGG**UU**ACA**U**GGGGA**U**GACCCCGCGAC**UU**GGA**UU**AGAG**U**C**U**C**UUUU**GGAA**U**AAGCC**U**GAA**U**GA**U**CCGAG**U**AGCA**U**C**U**CAG

List of all reading frames =

[**AUG**]

[**AUG** GGG AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]

[**AUG** ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]

[]

Are we done?

NO

4. Get reverse complement strand of DNA from given DNA:

AGCCATGTAGCTAACTCAGGTTACATGGGGATGACCCCGCGACTTGGATTAGAGTCTCTTTTGGAATAAGCCTGAATGATCCGAGTAGCATCTCAG

CTGAGATGCTACTCGGATCATTCAGGCTTATTCCAAAAGAGACTCTAATCCAAGTCGCGGGGTCATCCCCATGTAACCTGAGTTAGCTACATGGCT

5. Transcribe reverse complement DNA into RNA:

CTGAGATGCTACTCGGATCATTCAGGCTTATTCCAAAAGAGACTCTAATCCAAGTCGCGGGGTCATCCCCATGTAACCTGAGTTAGCTACATGGCT
CUGAGAUGCUACUCGGAUCAUUCAGGCUUAUUCCAAAAGAGACUCUAAUCCAAGUCGCGGGGUCAUCCCCAUGUAACCUGAGUUAGCUACAUGGCU

6. Count # of ‘AUG’ substrings:

CUGAGAUGCUACUCGGAUCAUUCAGGCUUAUUCCAAAAGAGACUCUAAUCCAAGUCGCGGGGUCAUCCCC AUGUAACCUGAGUUAGCUAC AUGGCU

7. Repeat the process from before adding to list of reading frames:

```
[AUG]
[AUG GGG AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]
[AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]
[]
[AUG CUA CUC GGA UCA UUC AGG CUU AUU CCA AAA GAG ACU CUA AUC CAA GUC GCG GGG UCA UCC CCA UGU
     AAC CUG AGU]
[AUG]
[]
```

8. Remove duplicates and empty lists

```
[AUG GGG AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]
[AUG ACC CCG CGA CUU GGA UUA GAG UCU CUU UUG GAA]
[AUG CUA CUC GGA UCA UUC AGG CUU AUU CCA AAA GAG ACU CUA AUC CAA GUC GCG GGG UCA UCC CCA UGU
     AAC CUG AGU]
[AUG]
```

9. Translate into protein

```
MGMTPRLGLESLLE
MTPRLGLESLLE
MLLGSFRLIPKETLIQVAGSSPCNLS
M
```

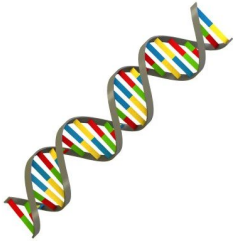
Partial Gener Ordering

Given: Positive integers n and k such that 100 >= n >= 0 and 10 >= k >= 0

Return: The total number of partial permutations P(n, k) modulo 1,000,000

We can compare the genomes by analyzing the ordering of their genes, then inferring which rearrangements have seperated the genes.

Enumerating Oriented Gene Orderings



Given: A positive integer n≤6.

Return: The total number of signed permutations of length n, followed by a list of all such permutations (you may list the signed permutations in any order).

Sample Input:

2

Sample Output:

```
8            < ---  # of permutations
-1 -2        < ---  list of permutations
-1 2
1 -2
1 2
-2 -1
-2 1
2 -1
 2 1
```

- Building off of “Enumerating Gene Orders”
- DNA has an orientation
  - RNA transcription only occurs in one direction
- Need to give synteny blocks an orientation to indicate which strand it is on
- Incorporating orientation of each index in permutation

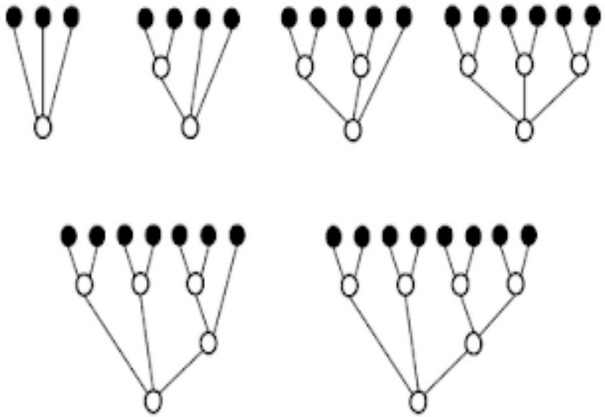
```python
import itertools

n = 3
permutation = []
x = 0
for i in itertools.permutations(list(range(1, n+1))):
    for j in itertools.product([-1,1],repeat = len(list(range(1, n+1)))):
        perms = [a*sign for a, sign in zip(i, j)]
        permutation.append(perms)
        x += 1
print(x)

for i in range(len(permutation)):
    print(*permutation[i], sep = ' ')
```

Counting Phylogenetic Ancestors

Given: A positive integer n (3≤n≤10000).

Return: The number of internal nodes of any unrooted binary tree having n leaves.

- Phylogenetic tree is a diagram that depicts the lines of evolutionary descent of different species, organisms, or genes from a common ancestor
  - Unrooted - does not depict their common ancestor
- Binary tree - each node has degree equal to at most 3
- Rooted tree - one node (the root) is set aside to serve as the pinnacle of the tree (and degree 2)
- Unrooted binary tree - all internal nodes have degree 3



Sample Input:
4

Sample Output:
2

Reversal Distance

- Permutation: Same as in ‘Enumerating Gene Order’, an ordered sequence of numbers to represent a region of DNA
- Reversal: a modification of a permutation made by flipping the order of some portion or portions of it
  - `Original: 1 2 3 4 5 6 7 8 9 10`
  - `Target: 1 3 2 4 5 10 9 8 7 6`
- Reversal distance: The minimum number of reversals required to turn the original permutation into the modified permutation
- Genome rearrangement: large scale mutations that affect entire intervals of DNA/RNA
- Most common form of mutation is inversion- basically just a reversal
- Reversals can be used to find sites of mutations and estimate their complexity
- Minimum number of inversions can be indicative of evolutionary distance between chromosomes

Reversal Distance

- Code solution utilizes Euna Park's 'Exact Greedy' algorithm
  - Master's Thesis from San Jose State University
- Greedy algorithm: prioritizes computational ease over finding the best answer immediately
  - Looks only for the best solution to a simple problem at each step
  - Combines these and compares combinations to determine best answer
- Exact meaning not an approximation
  - Euna also developed a few approximations that work faster but are less reliable
- Breakpoints: Locations where adjacency of numbers in the original and target sequences do not match
  - `Original: 1  2  3  4  5  6  7  8  9  10`
  - `Target:   1  3  2  4  5  10 9  8  7  6  .`
- Observations
  - Breakpoints occur at each end of a reversed substring
  - Breakpoints can occur at the beginning or end of a string
  - A reversal can never remove more than 2 breakpoints, but it can remove less
  - There is a direct relationship between the reversals formed and breakpoints
- Need for computerized solution: overlapping reversals
  - `Original: 1  2  3  4  5  6  7  8  9  10`
  - `Target:   1  3  4  2  5  10 9  6  7  8`
  - No obvious path between the two strings

Reversal Distance

Code implementation uses 4 functions:

First, a function to perform a reversal between indicated points on a sequence:

```
def reverse(sequence, start, end): #function to perform reversal between start and end indices
    pre=sequence[:start] #sequence up to start index
    revsec=sequence[start:end][::-1] #sequence in between start and end read in reverse
    post=sequence[end:] #sequence up to end index
    return pre+revsec+post #combining parts
```

Next, a function to find all the breakpoints between an original and target sequences:

```
def getbreakpoints(sequence, target): #function to find breakpoints between 2 sequences
    breakpts=[]
    for i in range(len(sequence)-1): #for each value in original sequence
        current=sequence[i] #store current value
        adjacent=sequence[i+1] #store next value
        if abs(target.index(current)-target.index(adjacent)) != 1: #if original sequence values aren't adjacent in target sequence
            breakpts.append(i+1) #add next value breakpoints list
    return breakpts #returns a list of all breakpoints between sequences
```

Reversal Distance

Code implementation uses 4 functions:

Thirdly, a function to find the minimum number of reversals between sequences:

```
def minimumreversals(sequences, target):
    revs=[] #create list to store reversals
    for seq in sequences:
        bkpts=getbreakpoints(seq, target) #use getbreapoints function
        for j in range(len(bkpts)-1): #iterate whole sequence over the number of breakpoints
            for k in range(j+1, len(bkpts)): #iterate from above to end of sequence as subsequences
                revs.append(reverse(seq, bkpts[j], bkpts[k])) #add reversals to list
    minbkpts=len(target) #minimum number of breakpoints is number of reversals performed
    minrev=[] #create list to store minimum number of reversals
    for rev in revs: #for each reversal
        numbkpts=len(getbreakpoints(rev, target)) #count number of breakpoints using function
        if numbkpts<minbkpts: #if the current number of breakpoints is less than previous min
            minbkpts=numbkpts #overwrite minimum number of breakpoints
            minrev=[rev] #overwrite minimum number of reversals
        elif numbkpts==minbkpts: #if the same number of breakpoints as previous min
            minrev.append(rev) #overwrite minimum number of reversals without changing minimum breakpoints
    return minrev #returns minimum number of required reversals to get to target from original sequence
```

Reversal Distance

Code implementation uses 4 functions:

Finally, a function that combines the 3 previous to calculate reversal distance using the observations made from looking at breakpoints:

```
def revdistance(sequence, target): #function that combines above functions to find reversal distance
    sequence=['-']+sequence+['+'] #formatting to allow combining of returns from other functions
    target=['-']+target+['+']
    revs=0 #counter to keep track of reversals used
    current=[sequence] #stores formatted sequence phrase in list to work with functions
    while target not in current: #until the two permutations match eachother:
        current=minimumreversals(current, target) #continue making reversals
        revs+=1 #adds one to counter for each reversal
    return revs
```

This last function is fed two sequences and returns reversal distance when printed:

```
#implementation of functions to solve provided dataset
print(revdistance(string[0], string[1]))
```

Rabbits and Recurrence Relations

- Fibonacci sequence is common example of a recurrence relation
- Recurrence relations are a way of finding terms in a sequence using the terms before
- Assumptions about rabbits:
  - Start with one pair of newborn rabbits
  - Takes one month to reach breeding age
  - Each month, every pair of breeding age rabbits breeds one litter of rabbits
  - No rabbits die within the provided timespan
  - In context of problem, consider only pairs of rabbits

Simple example to allow manual solution: 1 pair starting, 5 months total, 3 pairs per litter:

Month	1	2	3	4	5
Total Rabbit Pairs	1	1	4	7	19
Breeding Rabbit Pairs	0	1	1	4	7

[No. Breeding]=[Total of Prev. Month]

[Total Rabbits]=[Total of Prev. Month]+[No. of Breeding Prev. Month]*[Litter Size]

Rabbits and Recurrence Relations

```
1  string=open("rosalind_fib.txt").readlines() #import Rosalind file
2  split=string[0].split() #pull the first line from the file and store it in 'split'
3  months=int(split[0]) #store the n value in 'months'
4  littersize=int(split[1]) #store the k value in 'littersize'
5  lastmonthbreeding=0
6  lastmonthtotal=1
7  #'months' and 'littersize' are provided in Rosalind dataset
8  for i in range (months-1): #first month is accounted for as lastmonthtotal=1
9      currentbreeding=lastmonthtotal
10     currenttotal=lastmonthbreeding*littersize+lastmonthtotal
11     lastmonthbreeding=currentbreeding #currentbreeding now = lastmonthtotal
12     lastmonthtotal=currenttotal #update lastmonthtotal for next iteration
13 totalpairs=lastmonthtotal
14 print(totalpairs) #print result
```

Lines 1-4: Importing data from Rosalind download

Lines 5&6: First month will always be just 1 pair growing to breeding age, so accounted for here

Lines 8-12: Uses the rules found from manual solution and iterates over provided timespan

Lines 13&14: Remember, this result is pairs of rabbits- actual number is double that

GitHub: https://github.com/aburke921/Combinatorics.git

- Counting Phylogenetic Ancestors (INOD)
- Rabbits and Recurrence Relations(FIB)
- Mortal Fibonacci Rabbits (FIBD)
- Inferring mRNA from Protein (MRNA)
- Open Reading Frames (ORF)
- Enumerating Gene Orders (PERM)
- Enumerating Ordered Gene Orderings (SIGN)
- Reversal Distance (REAR)
- Partial Permutations (PPER)
- Introduction to Alternative Splicing (ASPC)
- Complementing a Strand of DNA (REVC)
- Transcribing DNA into RNA (RNA)
- Counting DNA Nucleotides (DNA)
- Mendel's First Law (IPRB)
- Overlap Graphs (GRPH)
- Completing a Tree (TREE)
- Computing GC Content (GC)
- Translating RNA into Protein (PROT)
- Longest Increasing Subsequence (LIGIS)
- Counting Point Mutations (HAMM)
- Finding a Shared Motif (LCSM)
- Finding a Motif in DNA (SUBS)
- Counting Subsets (SSET)
- RNA Splicing (SPLC)