Finding Hidden Patterns in DNA

- What makes searching for frequent subsequences hard?
    - Allowing for errors?
    - All the places they could be hiding?

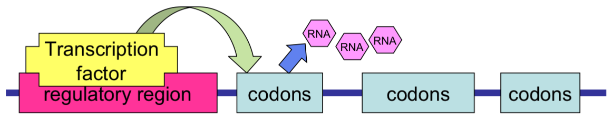
Hehheh They'll never find me this time
Doris was terrible at hide and seek

Initiating Transcription

- As a precursor to transcription (the reading of DNA to construct RNAs, that eventually leading to protein synthesis) special proteins bind to the DNA, and separate it to enable its reading.
- How do these proteins know where the coding genes are in order to bind?
- Genes are relatively rare
  - O(1,000,000,000) bases/genome
  - O(10000) genes/genome
  - O(1000) bases/gene
- Approximately 1% of DNA codes for genes ($10^3 10^4 / 10^9$)

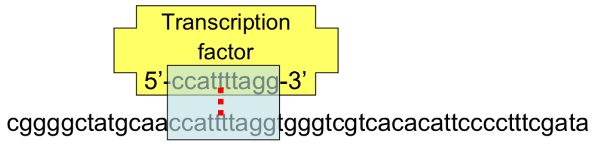
Regulatory Regions

- RNA polymerases seek out *regulatory* or *promoting* regions located 100-1000 bp upstream from the coding region
- They work in conjunction with special proteins called *transcription factors (TFs)* whose presence enables gene expression
- Within these regions are the *Transcription Factor Binding Sites (TFBS)*, special DNA sequence patterns known as *motifs* that are specific to a given transcription factor
- A Single TF can influence the expression of many genes. Through biological experiments one can infer, at least a subset of these affected genes.



Transcription Factor Binding Sites

- A TFBS can be located anywhere within the regulatory region.
- TFBS may vary slightly across different regulatory regions since non-essential bases could mutate
- Transcription factors are robust (they will still bind) in the presence of small sequence differences by a few bases



Identifying Motifs: Complications

- We don’t know the motif sequence for every TF
- We don’t know where it is located relative to a gene’s start
- Moreover, motifs can differ slightly from one gene to the next
- We *only* know that it occurs somewhere near genes that share a TF
- How to discern a Motif’s frequent *similar* pattern from *random* patterns?
- How is this problem different that finding frequent k-mers from Lecture 2?

Let's look for an *Easy* Motif

```
 1 tagtggtcttttgagtgtagatccgaagggaaagtatttccaccagttcggggtcacccagcagggcagggtgacttaat
 2 cgcgactcggcgctcacagttatcgcacgtttagaccaaaacggagttagatccgaaactggagtttaatcggagtcctt
 3 gttacttgtgagcctggttagatccgaaatataattgttggctgcatagcggagctgacatacgagtaggggaaatgcgt
 4 aacatcaggctttgattaaacaatttaagcacgtagatccgaattgacctgatgacaatacggaacatgccggctccggg
 5 accaccggataggctgcttattagatccgaaaggtagtatcgtaataatggctcagccatgtcaatgtgcggcattccac
 6 tagatccgaatcgatcgtgtttctccctctgtgggttaacgaggggtccgaccttgctcgcatgtgccgaacttgtaccc
 7 gaaatggttcggtgcgatatcaggccgttctcttaacttggcggtgtagatccgaacgtctctggaggggtcgtgcgcta
 8 atgtatactagacattctaacgctcgcttattggcggagaccatttgctccactacaagaggctactgtgtagatccgaa
 9 ttcttacacccttctttagatccgaacctgttggcgccatcttcttttcgagtccttgtacctccatttgctctgatgac
10 ctacctatgtaaaacaacatctactaacgtagtccggtctttcctgatctgccctaacctacaggtagatccgaaattcg
```

Problem: Given *M* sequences of length *N* find any *k*-mer that appears in each sequence.

How would you go about finding a 10-mer that appears in *every one* of these strings?

Sneak Peek at the Answer

```
 1 tagtggtcttttgagtgTAGATCCGAAgggaaagtatttccaccagttcggggtcacccagcagggcagggtgacttaat
 2 cgcgactcggcgctcacagttatcgcacgtttagaccaaaacggagtTAGATCCGAAactggagtttaatcggagtcctt
 3 gttacttgtgagcctggtTAGATCCGAAatataattgttggctgcatagcggagctgacatacgagtaggggaaatgcgt
 4 aacatcaggctttgattaaacaatttaagcacgTAGATCCGAAttgacctgatgacaatacggaacatgccggctccggg
 5 accaccggataggctgcttatTAGATCCGAAaggtagtatcgtaataatggctcagccatgtcaatgtgcggcattccac
 6 TAGATCCGAAtcgatcgtgtttctccctctgtgggttaacgaggggtccgaccttgctcgcatgtgccgaacttgtaccc
 7 gaaatggttcggtgcgatatcaggccgttctcttaacttggcggtgTAGATCCGAAcgtctctggaggggtcgtgcgcta
 8 atgtatactagacattctaacgctcgcttattggcggagaccatttgctccactacaagaggctactgtgTAGATCCGAA
 9 ttcttacacccttcttTAGATCCGAAcctgttggcgccatcttcttttcgagtccttgtacctccatttgctctgatgac
10 ctacctatgtaaaacaacatctactaacgtagtccggtctttcctgatctgccctaacctacaggTAGATCCGAAattcg
```

Now that you've seen the answer, how would you find it?

Meet Mr *Brute Force*

- He's often the best starting point when approaching a problem
- He'll also serve as a straw-man when designing new approaches
- Though he's seldom elegant, he gets the job done
- Often, we can't afford to wait for him

For our current problem a brute force solution would consider every k-mer position in all strings and see if they match. Given M sequences of length N, there are:

$$(N - k + 1)^M$$

position combinations to consider.

How do you write *M* nested loops when *M* is a variable?

A Library of Helper Functions

- There's a tendancy to approach this problem with a series of nested for-loops, while the approach is valid, it doesn't generalize. It assumes a specific number of sequences.
- What we need is an *iterator* that generates all permutations of a sequence.
- This nested-for-loop iterator is called a *Cartesian Product* over sets.
- Python has a library to accomplish this

Using *itertools*

```python
import itertools

for number in itertools.product("01", repeat=3):
    print ''.join(number)
```

```
000
001
010
011
100
101
110
111
```

All permutations of items from a list

```
N = 0
for number in itertools.product(range(3), repeat=3):
    print number,
    N += 1
    if (N % 5 == 0):
        print
```

```
(0, 0, 0) (0, 0, 1) (0, 0, 2) (0, 1, 0) (0, 1, 1)
(0, 1, 2) (0, 2, 0) (0, 2, 1) (0, 2, 2) (1, 0, 0)
(1, 0, 1) (1, 0, 2) (1, 1, 0) (1, 1, 1) (1, 1, 2)
(1, 2, 0) (1, 2, 1) (1, 2, 2) (2, 0, 0) (2, 0, 1)
(2, 0, 2) (2, 1, 0) (2, 1, 1) (2, 1, 2) (2, 2, 0)
(2, 2, 1) (2, 2, 2)
```

Permutations of mixed types

```
for section in itertools.product(("I", "II", "III", "IV"),"ABC",range(1,3)):
    print section
```

```
('I', 'A', 1)
('I', 'A', 2)
('I', 'B', 1)
('I', 'B', 2)
('I', 'C', 1)
('I', 'C', 2)
('II', 'A', 1)
('II', 'A', 2)
('II', 'B', 1)
('II', 'B', 2)
('II', 'C', 1)
('II', 'C', 2)
('III', 'A', 1)
('III', 'A', 2)
('III', 'B', 1)
('III', 'B', 2)
('III', 'C', 1)
('III', 'C', 2)
('IV', 'A', 1)
('IV', 'A', 2)
('IV', 'B', 1)
('IV', 'B', 2)
('IV', 'C', 1)
('IV', 'C', 2)
```

Now let's try some *Brute Force* code

```
sequences = [
    'tagtggtcttttgagtgtagatccgaagggaaagtatttccaccagttcggggtcacccagcagggcagggtgacttaat',
    'cgcgactcggcgctcacagttatcgcacgtttagaccaaaacggagttagatccgaaactggagtttaatcggagtcctt',
    'gttacttgtgagcctggttagatccgaaatataattgttggctgcatagcggagctgacatacgagtaggggaaatgcgt',
    'aacatcaggctttgattaaacaatttaagcacgtagatccgaattgacctgatgacaatacggaacatgccggctccggg',
    'accaccggataggctgcttattagatccgaaaggtagtatcgtaataatggctcagccatgtcaatgtgcggcattccac',
    'tagatccgaatcgatcgtgtttctccctctgtgggttaacgaggggtccgaccttgctcgcatgtgccgaacttgtaccc',
    'gaaatggttcggtgcgatatcaggccgttctcttaacttggcggtgtagatccgaacgtctctggaggggtcgtgcgcta',
    'atgtatactagacattctaacgctcgcttattggcggagaccatttgctccactacaagaggctactgtgtagatccgaa',
    'ttcttacacccttctttagatccgaacctgttggcgccatcttcttttcgagtccttgtacctccatttgctctgatgac',
    'ctacctatgtaaaacaacatctactaacgtagtccggtctttcctgatctgccctaacctacaggtagatccgaaattcg']

def bruteForce(dna,k):
    """Finds a *k*-mer common to all sequences from a
       list of *dna* fragments with the same length"""
    M = len(dna)     # how many sequences
    N = len(dna[0])  # length of sequences
    for offset in itertools.product(range(N-k+1), repeat=M):
        for i in xrange(1,len(offset)):
            if dna[0][offset[0]:offset[0]+k] != dna[i][offset[i]:offset[i]+k]:
                break
        else:
            return offset, dna[0][offset[0]:offset[0]+10]
```

Test and then time it

```
M = 4
position, motif = bruteForce(sequences[0:M], 10)
print position, motif
print
for i in xrange(M):
    p = position[i]
    print sequences[i][:p]+sequences[i][p:p+10].upper()+sequences[i][p+10:]
print

%timeit bruteForce(sequences[0:M], 10)
# you can try a larger value of M, but be prepared to wait
```

```
(17, 47, 18, 33) tagatccgaa

tagtggtcttttgagtgTAGATCCGAAgggaaagtatttccaccagttcggggtcacccagcagggcagggtgacttaat
cgcgactcggcgctcacagttatcgcacgtttagaccaaaacggagtTAGATCCGAAactggagtttaatcggagtcctt
gttacttgtgagcctggtTAGATCCGAAatataattgttggctgcatagcggagctgacatacgagtaggggaaatgcgt
aacatcaggctttgattaaacaatttaagcacgTAGATCCGAAttgacctgatgacaatacggaacatgccggctccggg

1 loop, best of 3: 4.74 s per loop
```

Approximate Matching

Now let's consider a more realistic motif finding problem, where the binding sites do not need to match exactly.

```
 1 tagtggtcttttgagtgTAGATCTGAAgggaaagtatttccaccagttcggggtcacccagcagggcagggtgacttaat
 2 cgcgactcggcgctcacagttatcgcacgtttagaccaaaacggagtTGGATCCGAAactggagtttaatcggagtcctt
 3 gttacttgtgagcctggtTAGACCCGAAatataattgttggctgcatagcggagctgacatacgagtaggggaaatgcgt
 4 aacatcaggctttgattaaacaatttaagcacgTAAATCCGAAttgacctgatgacaatacggaacatgccggctccggg
 5 accaccggataggctgcttatTAGGTCCAAAaggtagtatcgtaataatggctcagccatgtcaatgtgcggcattccac
 6 TAGATTCGAAtcgatcgtgtttctccctctgtgggttaacgaggggtccgaccttgctcgcatgtgccgaacttgtaccc
 7 gaaatggttcggtgcgatatcaggccgttctcttaacttggcggtgCAGATCCGAAcgtctctggaggggtcgtgcgcta
 8 atgtatactagacattctaacgctcgcttattggcggagaccatttgctccactacaagaggctactgtgTAGATCCGTA
 9 ttcttacacccttcttTAGATCCAAAcctgttggcgccatcttcttttcgagtccttgtacctccatttgctctgatgac
10 ctacctatgtaaaacaacatctactaacgtagtccggtctttcctgatctgccctaacctacaggTCGATCCGAAattcg
```

Actually, none of the sequences have an unmodified copy of the original motif

Profile and Consensus

How to find approximate string matches?

- Align candidate motifs by their start indexes
  $$s = (s_1, s_2, \ldots, s_t)$$
- Construct a matrix profile with the frequencies of each nucleotide in columns
- Consensus nucleotide in each position has the highest score in each column

```
                a G g t a c T t
                C c A t a c g t
Alignment       a c g t T A g t
                a c g t C c A t
                C c g t a c g G

              _________________

              A 3 0 1 0 3 1 1 0
Profile       C 2 4 0 0 1 4 0 0
              G 0 1 4 0 0 0 3 1
              T 0 0 0 5 1 0 1 4
              _________________
Consensus       A C G T A C G T
```

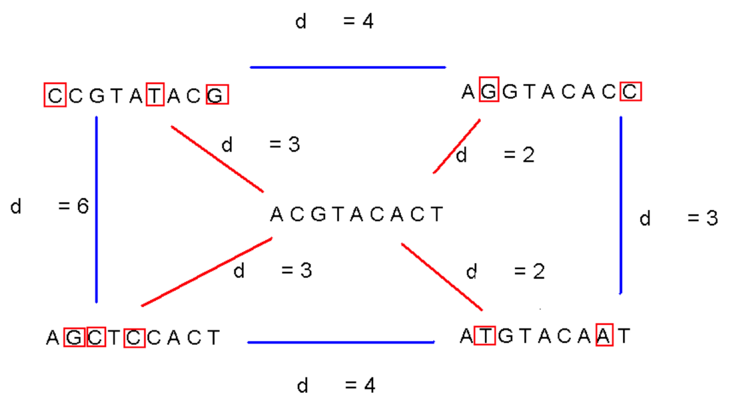
Consensus

- One can think of the consensus as an ***ancestor*** motif, from which mutated motifs emerged
- The *distance* between an actual motif and the consensus sequence is generally less than that for any two actual motifs
- *Hamming distance* is number of positions that differ between two strings

```
G A G A C T C A T
X         X
T A G A C G C A T
```

Consensus Properties

- A consensus string has a minimal hamming distance to all its source strings

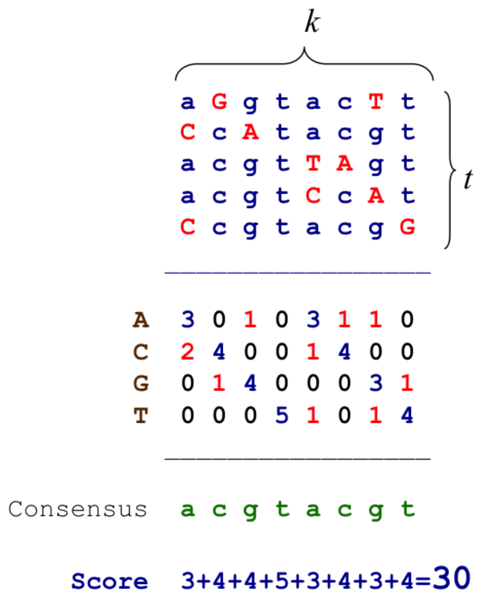


Scoring Motifs

- Given $s = (s_1, s_2, \ldots, s_t)$ and *DNA*

$$Score(s, DNA) = \sum_{i=1}^{k} \max_{j \in A,C,G,T} count(j, i)$$

- So our approach is back to *brute force*
  - We consider every candidate motif in every string
  - Return the set of indices with the highest score



```
a G g t a c T t
C c A t a c g t
a c g t T A g t
a c g t C c A t
C c g t a c g G
```

A	3	0	1	0	3	1	1	0
C	2	4	0	0	1	4	0	0
G	0	1	4	0	0	0	3	1
T	0	0	0	5	1	0	1	4

Consensus a c g t a c g t

Score 3+4+4+5+3+4+3+4=30

Let's try again, and handle errors this time!

```
def Score(s, DNA, k):
    """
        compute the consensus SCORE of a given k-mer alignment given
        offsets into each DNA string. s = list of starting indices.
        DNA = list of nucleotide strings. k = Target Motif length
    """
    score = 0
    for i in xrange(k):
        # loop over string positions
        cnt = dict(zip("acgt",(0,0,0,0)))
        for j, sval in enumerate(s):
            base = DNA[j][sval+i]
            cnt[base] += 1
        score += max(cnt.values())
    return score

def BruteForceMotifSearch(dna,k):
    M = len(dna)     # how many sequences
    N = len(dna[0])  # length of sequences
    bestScore = 0
    bestAlignment = []
    for offset in itertools.product(range(N-k+1), repeat=M):
        s = Score(offset,dna,k)
        if (s > bestScore):
            bestAlignment = [p for p in offset]
            bestScore = s
    print bestAlignment, bestScore
```

Test and time

```
seqApprox = [
    'tagtggtcttttgagtgtagatctgaagggaaagtatttccaccagttcggggtcacccagcagggcagggtgacttaat',
    'cgcgactcggcgctcacagttatcgcacgtttagaccaaaacggagttggatccgaaactggagtttaatcggagtcctt',
    'gttacttgtgagcctggttagacccgaaatataattgttggctgcatagcggagctgacatacgagtaggggaaatgcgt',
    'aacatcaggctttgattaaacaatttaagcacgtaaatccgaattgacctgatgacaatacggaacatgccggctccggg',
    'accaccggataggctgcttattaggtccaaaaggtagtatcgtaataatggctcagccatgtcaatgtgcggcattccac',
    'tagattcgaatcgatcgtgtttctccctctgtgggttaacgaggggtccgaccttgctcgcatgtgccgaacttgtaccc',
    'gaaatggttcggtgcgatatcaggccgttctcttaacttggcggtgcagatccgaacgtctctggaggggtcgtgcgcta',
    'atgtatactagacattctaacgctcgcttattggcggagaccatttgctccactacaagaggctactgtgtagatccgta',
    'ttcttacacccttctttagatccaaacctgttggcgccatcttcttttcgagtccttgtacctccatttgctctgatgac',
    'ctacctatgtaaaacaacatctactaacgtagtccggtctttcctgatctgccctaacctacaggtcgatccgaaattcg']

%timeit Score([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], seqApprox, 10)
%time BruteForceMotifSearch(seqApprox[0:4], 10)
```

```
10000 loops, best of 3: 40.6 µs per loop
[17, 47, 18, 33] 36
CPU times: user 17min 22s, sys: 1.97 s, total: 17min 24s
Wall time: 17min 24s
```

Running Time of BruteForceMotifSearch

- Search *(N - k + 1)* positions in each of *M* sequences, by examining $(N - k + 1)^M$ sets of starting positions
- For each set of starting positions, the scoring function makes *O(Mk)* operations, so complexity is

$$Mk(N-k+1)^M = O(MkN^M)$$

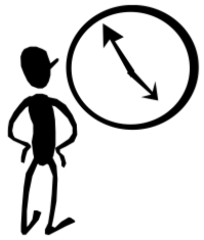
- That means that for *M = 10, N = 80, k = 10* we must perform approximately 10^{21} computations
- Generously assuming 10^9 comps/sec it will require only 10^{12} secs

$\frac{10^{12}}{(60*60*24*365)} > 30000$ years

- Want to wait?

How conservative is this estimate?

- For the example we just did *M = 4, N = 80, k = 10*
- So that gives $\approx 4.0 \times 10^9$ operations
- Using our 10^9 operations per second estimate, it should have taken ***only 4 secs***.
- Instead it took closer to 700 secs, which suggests we are getting around 5.85 million operations per second.
- So, in reality it will even take longer!



How can we find Motifs in our lifetime?

- Should we give up on Python and write in C? Assembly Language?
- Will biological insights save us this time?
- Are there other ways to find Motifs?

- Consider that if you knew what motif you were looking for, it would take only

$$k(N - k + 1)M = O(kNM)$$

- Is that significantly better?