Finding TFBS Motifs in our Lifetime

- Recall from last time that the *Brute Force* approach for finding a common 10-mer motif common to 10 sequences of length 80 bases was going to take up roughly 30,000 years
- Today we will consider alternative and non-obvious approaches for solving this problem
- We will trade one old man (us) for another (an Oracle)
Recall from last Lecture

- The following set of 10 sequences have an embedded noisy motif, \textit{TAGATCCGAA}.

1. There are no exact matches
2. The consensus motif gives a good score
Consensus Scoring Function

- We developed a consensus scoring function to address noise
- But, we needed to apply it an exponential number, $O(N^M)$ of times!
- Here's the scoring function...

```python
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, 1-based, 0 means ignore
    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):
            # loop over DNA strands
            base = DNA[j][sval+i]
            cnt[base] += 1
        score += max(cnt.itervalues())
    return score
```
And here's the Score we're looking for...

```python
seqApprox = [
    'tagtggttttgagttagatctgaaggaagcgatattttcaccagttcggtgtcaccagcaggcaggtgacttaat',
    'cgcgacgccgctcagttatcgcacgtagtacgatccgaacactggagttaatcggagtcctc',
    'gttacttttgagacctgtgttagaccccaaataaatgtggctgtcatagcagagtagtagggaatatcgt',
    'aactacgagcttttgaattacaacatccaatatgggaaggatactcgtgtgacacatcgtgagttggag',
    'accaccggataggtgctttattaggtccaaaaagttagatcgttaataatggtctagcagccatgtcaatg',
    'tagattcgaatcagcttggtttctccctctgttgtgaaaccagggttcgacactcttgctgtgacgcaacct',
    'gaaatggttcgccgtgatcagctggctttcttttaacttgccggtcacatcggaaacgctcgcgtgctc',
    'atgtatactagacattctcaacgtcgtttattggtcggagaccatttttgcacactacaagaggctactgtgtgatgc',
    'ttcctacaccccttttagatccaaacctctgtggtgcacactctctctttcactgtgtgctgtgatgc',
    'ctacctagtaaaacaacctataacgtatcgcagctccggctcttctgtctcatgtgccccataacctacaggtcagttccaaatcgt']
```

```
print Score([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], seqApprox, 10)
89
```

```
%timeit Score([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], seqApprox, 10)
10000 loops, best of 3: 39.9 µs per loop
```

So even at a blazing 40µs we'll need many lifetimes to compute the 70^{10} scores
Pruning Trees

• One method for reducing the computational cost of a search algorithm is to prune the space of permutations that could not possibly lead to a better answer than the current best answer.

• Pruning decisions are based on solutions to subproblems that appear early on and offer no hope.

• How does this apply to our Motif finding problem?

Consider any permutation of offsets that begins with the indices [25, 63, 10, 43, ...]. Just based on the first 4 indices the largest possible score is $17 + 6 \times 10 = 87$, which assumes that all 6 remaining strings match perfectly at all 10 positions.

If the best answer so far is 89, there is no need to consider the $70^6$ offset permutations that start with these 4 indices.
Our standard method for enumerating permutations can be considered as a traversal of leaf nodes in a search tree.

Suppose after checking the first few offsets could know already that any score of children nodes could not beat the best score seen so far?
Branch-and-Bound Motif Search

- Since each level of the tree goes deeper into search, discarding a prefix discards all following branches
- This saves us from looking at \((N-k+1)^{t-depth}\) leaves
- Note our enumeration of tree-branches is depth-first
- We’ll formulate of trimming algorithm as a recursive algorithm
A Recursive Exploration of a Search Tree

```python
bestAlignment = []
prunedPaths = 0

def exploreMotifs(DNA, k, path, bestScore):
    """ Search for a k-length motif in the list of DNA sequences by exploring all paths in a search tree. Each call extends path by one. Once the path reaches the number of DNA strings a score is computed. """
    global bestAlignment, prunedPaths
    depth = len(path)
    M = len(DNA)
    if (depth == M):  # here we have an index in all M sequences
        s = Score(path, DNA, k)
        if (s > bestScore):
            bestAlignment = [p for p in path]
        return s
    else:
        return bestScore

else:
    # Let's consider if an optimistic best score can beat the best score so far
    if (depth > 1):
        OptimisticScore = k*(M-depth) + Score(path, DNA, k)
    else:
        OptimisticScore = k*M
    if (OptimisticScore < bestScore):
```

[17, 47, 18, 33, 21, 0] 53 8615931
CPU times: user 5min 17s, sys: 533 ms, total: 5min 17s
Wall time: 5min 17s
Observations

- For our problem instance, Branch-and-Bound Motif finding is significantly faster
  - It found a motif in the first 6 strings in less time than the Brute Force approach found a solution in the first 4 strings
  - More than $70^2 \approx 5000$ times faster
  - It did so by trimming more than 8 Million paths
  - Trimming added extra calls to Score (basically doubling the worst-case number of calls), but ended up saving even more hopeless calls along longer paths.
  - In practice, Branch-and-Bound, significantly improved the average performance
- Does this improve the worst-case performance from $O(kN^M)$?
  - What if all of our motifs were found at the end of each DNA string?
  - How do we avoid these worse case data sets?
  - Randomize the search-tree tranversal order

$\approx$
We need a new approach

- Enumerating every possible permutation of motif positions is still not getting us the speed we want.
- Let's try another tried-and-tested approach to algorithm design, mixing up the problem
  - Suppose that some Oracle could tell us what the motif is
  - How long would it take us to find its position in each string?
  - We could compute the Hamming Distance from our given motif to the k-mer at every position of each DNA sequence and keep track of the smallest distance and its position on each string.
  - These positions are our best guess of where the motif can be found on each string
- Let's call this approach scanning-and-scoring to find a given motif.
Scanning-and-Scoring a Motif

```python
def ScanAndScoreMotif(DNA, motif):
    totalDist = 0
    bestAlignment = []
    k = len(motif)
    for seq in DNA:
        minHammingDist = k+1
        for s in xrange(len(seq)-k+1):
            HammingDist = sum([1 for i in xrange(k) if motif[i] != seq[s+i]])
            if (HammingDist < minHammingDist):
                bestS = s
                minHammingDist = HammingDist
                bestAlignment.append(bestS)
                totalDist += minHammingDist
    return bestAlignment, totalDist

print ScanAndScoreMotif(seqApprox, "tagatccgaa")
%timeit ScanAndScoreMotif(seqApprox, "tagatccgaa")

([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11)
1000 loops, best of 3: 1.41 ms per loop

Wow, we can test over 650 motifs per second!
```
Scan-and-Score Motif Performance

- There are $M(N - k + 1)$ positions to test the motif, and each test requires $k$ tests.
- So each scan is $O(MNk)$.

- So where where do we get candidate motifs?
- Can we try all of them? There are $4^{10} = 1048576$ in our example.
  - Do the math, $1048576$ motifs $\times 2$ mS $\approx 35$ mins
  - Not fast, but less than a lifetime

- This approach is called a **Median String Motif Search**
- Recall from last Lecture that a string that minimizes Hamming distance is like finding a middle or median string that is closer to all instances than the instances are to each other.
Let's Do It

```python
import itertools

def MedianStringMotifSearch(DNA, k):
    """ Consider all possible 4**k motifs""
    bestAlignment = []
    minHammingDist = k*len(DNA)
    kmer = ''
    for pattern in itertools.product('acgt', repeat=k):
        motif = ''.join(pattern)
        align, dist = ScanAndScoreMotif(DNA, motif)
        if (dist < minHammingDist):
            bestAlignment = [p for p in align]
            minHammingDist = dist
            kmer = motif
    return bestAlignment, minHammingDist, kmer

%time MedianStringMotifSearch(seqApprox, 10)

CPU times: user 26min 35s, sys: 613 ms, total: 26min 35s
Wall time: 26min 35s

([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')

Should we declare victory and move on? Do you find anything uncomfortable about an algorithm that requires $O(MNk^4)$ steps?
Notes on Median String Motif Search

- Similarities between finding and alignment with minimal Hamming Distance and maximizing a Motif's consensus score.
- In fact, if instead of counting mismatches as in the code fragment:
  \[
  \text{HammingDist} = \sum([1 \text{ for } i \text{ in xrange(k)} \text{ if motif}[i] \neq \text{seq}[s+i]])
  \]
  
  we had counted matches
  \[
  \text{Matches} = \sum([1 \text{ for } i \text{ in xrange(k)} \text{ if motif}[i] = \text{seq}[s+i]])
  \]
  
  and found the \textit{maximum(TotalMatches)} instead of the \textit{min(TotalHammingDistance)} we would be using the same measure as \textit{Score()}.
- Thus, we expect \textit{MedianStringMotifSearch()} to give the same answer as either \textit{BruteForceMotifSearch()} or \textit{BranchAndBoundMotifSearch()}.  
- However, the $4^k$ term raises some concerns. If $k$ were instead 20, then we'd have to Scan-and-Score more than $10^{12}$ times. Another \textit{not-in-a-lifetime} algorithm.
- We can also apply the \textit{Branch-and-Bound} approach to the Median string method, but, as before it would only improve the average case.
Other ways to guess the motif?

- If we knew that the motif that we are looking for was contained somewhere in our DNA sequences we could test the \((N - k + 1)t\) motifs from our DNA, giving a \(O(N^2 t^2)\) algorithm.

- Unfortunately, as you may recall our motif did not appear actually appear in our data.

- You could keep track of a few good motif candidates using a manageable and perhaps random subsets of the given DNA sequences, and use them as your candidate motifs.
Let's try considering only Motifs seen in the DNA

```python
def ContainedMotifSearch(DNA, k):
    """ Consider only motifs from the given DNA sequences"""
    motifSet = set()
    for seq in DNA:
        for i in xrange(len(seq)-k+1):
            motifSet.add(seq[i:i+k])
    print "%d Motifs in our set" % len(motifSet)
    bestAlignment = []
    minHammingDist = k*len(DNA)
    kmer = ''
    for motif in motifSet:
        align, dist = ScanAndScoreMotif(DNA, motif)
        if (dist < minHammingDist):
            bestAlignment = [s for s in align]
            minHammingDist = dist
            kmer = motif
    return bestAlignment, minHammingDist, kmer

%time ContainedMotifSearch(seqApprox, 10)

709 Motifs in our set
CPU times: user 1.33 s, sys: 16 ms, total: 1.34 s
Wall time: 1.33 s

([17, 31, 18, 33, 21, 0, 46, 70, 16, 65], 17, 'tagatccaaa')

Not exactly the motif we were looking for (off by a 'g'),   [17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa', but boy was it fast! Where's a good Oracle when you need one?
```
Insights from the consensus score matrix

If we call Score([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], seqApprox, 10)

DNA[0][17:27]  
DNA[1][31:41]  
DNA[2][18:28]  
DNA[3][33:43]  
DNA[4][21:31]  
DNA[5][0:10]  
DNA[6][46:56]  
DNA[7][70:80]  
DNA[8][16:26]  
DNA[9][65:75]  

a [0, 9, 1, 9, 0, 0, 1, 3, 9, 10]  
c [1, 1, 0, 0, 2, 9, 8, 0, 0, 0]  
g [0, 0, 9, 1, 0, 0, 0, 7, 0, 0]  
t [9, 0, 0, 0, 8, 1, 1, 0, 1, 0]  

[9, 9, 9, 9, 8, 9, 8, 7, 9, 10]  
Score = 87  

Consensus  
t a g a t c c g a a  

Any Ideas?
Contained Consensus Motif Search

```python
def Consensus(s, DNA, k):
    """ compute the consensus k-Motif of an alignment given offsets into each DNA string.
    s = list of starting indices, 1-based, 0 means ignore, DNA = list of nucleotide strings,
    k = Target Motif length """
    consensus = ''
    for i in xrange(k):
        # loop over string positions
        cnt = dict(zip("acgt",(0,0,0,0)))
        for j, sval in enumerate(s):
            # loop over DNA strands
            base = DNA[j][sval+i]
            cnt[base] += 1
            consensus += max(cnt.items(), key=lambda tup: tup[1][0])
    return consensus

def ContainedConsensusMotifSearch(DNA, k):
    bestAlignment, minHammingDist, kmer = ContainedMotifSearch(DNA, k)
    motif = Consensus(bestAlignment, DNA, k)
    newAlignment, HammingDist = ScanAndScoreMotif(DNA, motif)
    return newAlignment, HammingDist, motif

%time ContainedConsensusMotifSearch(seqApprox, 10)
```

709 Motifs in our set
CPU times: user 1.06 s, sys: 23 ms, total: 1.08 s
Wall time: 1.06 s

([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')

Now we're cooking!
Dad, are we there yet?

We got the answer that we were looking for, but

- How can we be sure it will always give the correct answer?
  - Our other methods were exhaustive, they examined every possibility
  - This method considers only a subset of solutions, picks the best one in a greedy fashion
  - What if there had been ties among the candidate motifs?
  - What if the consensus score (87% matches) had been lower
  - Would we, should we, be satisfied?

- It's one thing to be greedy, and another to be both greedy and biased
  - Our method is greedy in that it considers only the best contained motif, greedy methods are subject to falling into local minimums
  - Since consider only subsequences as motifs we introduce bias

- Note that Consensus can generate motifs not seen in our data
A randomized approach to motif finding

- One way to avoid bias and local minima is to introduce *randomness*
- We can generate candidate motifs from our data by treating it as distribution
  - likely motif candidates from this distribution are those generated by Consensus
  - Consensus strings can be tested by Scan-and-Score and their alignments lead to new consensus strings
  - Eventually, we should converge to some local minimal answer
- To avoid finding a local minimum, we try several random starts, and search for the best score amongst all these starts.
- A randomized algorithm does not guarantee an optimal solution. Instead it promises a good/plausible answer on average, and it is not susceptible to a worse-case data sets as our greedy/biased method was.
import random

def RandomizedMotifSearch(DNA, k):
    """ Searches for a k-length motif that appears in all given DNA sequences. It begins with a random set of candidate consensus motifs derived from the data. It refines the motif until a true consensus emerges."""

    # Seed motifs from random alignments
    motifSet = set()
    for i in xrange(500):
        randomAlignment = [random.randint(0, len(DNA[j]) - k) for j in xrange(len(DNA))]
        motif = Consensus(randomAlignment, DNA, k)
        motifSet.add(motif)

    bestAlignment = []
    minHammingDist = k * len(DNA)
    kmer = ''
    testSet = motifSet.copy()
    while len(testSet) > 0:
        print len(motifSet),
        nextSet = set()
        for motif in testSet:
            align, dist = ScanAndScoreMotif(DNA, motif)
            # add new motifs based on these alignments
Let's try it

```python
%time RandomizedMotifSearch(seqApprox,10)
```

```bash
500 749 822 839 842CPU times: user 1.43 s, sys: 23 ms, total: 1.45 s
Wall time: 1.56 s

([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')
```

Randomized algorithms should be restarted multiple times to insure a stable solution.

```python
for i in xrange(10):
    print RandomizedMotifSearch(seqApprox,10)
```

```bash
500 751 820 836 837 ([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')
500 750 825 838 844 ([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')
500 755 837 856 859 ([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')
499 745 814 831 834 ([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')
500 760 837 859 862 863 864 ([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')
500 744 813 825 827 ([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')
498 746 830 846 850 851 ([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')
500 766 848 864 866 ([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')
500 728 800 810 811 ([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')
500 750 833 851 852 ([17, 47, 18, 33, 21, 0, 46, 70, 16, 65], 11, 'tagatccgaa')
```
Lessons Learned

- We can find Motifs in our lifetime
  - Practical exhaustive search algorithm for small $k$, MedianStringMotifSearch()
  - Practical fast algorithm RandomizedMotifSearch(DNA, $k$)
- Three algorithm design approaches "Branch-and-Bound", "Greedy", and "Randomized"
- Reversing the objective, pretending that you know the answer, and validating it
- The power of randomness
  - Not susceptible to worse case data
  - Avoids local minimums that plague some greedy algorithms