Randomized Algorithms

- **Final Exam**
  - Friday, May 1
  - (8am-11am)

- **Final Study Session**
  - Monday, April 27
  - (4pm-6pm)
Randomized Algorithms

- Randomized algorithms incorporate random, rather than deterministic, decisions.
- Commonly used in situations where no exact and/or fast algorithm is known.
- Works for algorithms that behave well on typical data, but poorly in special cases.
- Main advantage is that no input can reliably produce worst-case results because the algorithm runs differently each time.
Select

• **Select(L, k)** finds the k\(^{th}\) smallest element in L
• Select(L,1) find the smallest…
  – Well known O(n) algorithm
    
    \[
    \text{minv} = \text{HUGE} \\
    \text{for } v \text{ in } L: \\
    \quad \text{if } (v < \text{minv}): \\
    \quad \quad \text{minv} = v
    \]

• Select(L, \text{len}(L)/2) find the median…
  – How?
    – median = \text{sorted(L)}[\text{len}(L)/2] \quad \text{O(n logn)}
• Can we find medians, or 1\(^{st}\) quartiles in O(n)?
Select Recursion

- **Select(L, k)** finds the $k^{th}$ smallest element in **L**
  - Select an element $m$ from unsorted list **L** and partition **L** the array into two smaller lists:
    
    $L_{lo}$ - elements smaller than $m$

    and

    $L_{hi}$ - elements larger than $m$

    if ($\text{len}(L_{lo}) \geq k$) then
      Select($L_{lo}$, k)
    
    elif ($k > \text{len}(L_{lo}) + 1$) then
      Select($L_{hi}$, $k - (\text{len}(L_{lo}) + 1)$)
    
    else $m$ is the $k^{th}$ smallest element
Example of Select(L, 5)

Given an array: \( L = \{ 6, 3, 2, 8, 4, 5, 1, 7, 0, 9 \} \)

**Step 1:** Choose the first element as \( m \)

\[
L = \{ 6, 3, 2, 8, 4, 5, 1, 7, 0, 9 \}
\]
Step 2: Split the array into $L_{lo}$ and $L_{hi}$

$L_{lo} = \{ 3, 2, 4, 5, 1, 0 \}$

$L = \{ 6, 3, 2, 8, 4, 5, 1, 7, 0, 9 \}$

$L_{hi} = \{ 8, 7, 9 \}$
Example of Select(L,5) (cont’d)

**Step 3:** Recursively call Select on either $L_{lo}$ or $L_{hi}$ until $\text{len}(L_{lo})+1 = k$, then return $m$.

\[
\text{len}(L_{lo}) > k = 5 \quad \square \quad \text{Select}([3, 2, 4, 5, 1, 0], 5) \quad m = 3
\]

\[
L_{lo} = [2, 1, 0] \quad L_{hi} = [4, 5]
\]

\[
k = 5 > \text{len}(L_{lo}) + 1 \quad \square \quad \text{Select}([4, 5], 5 - 3 - 1) \quad m = 4
\]

\[
L_{lo} = \text{empty}, \quad L_{hi} = [5]
\]

\[
k = 1 == \text{len}(L_{lo}) + 1 \quad \square \quad \text{return} \quad 4
\]
Select Code

```python
In [47]:
def select(L, k):
    value = L[0]
    Llo = [t for t in L if t < value]
    Lhi = [t for t in L if t > value]
    below = len(Llo) + 1
    if (len(Llo) >= k):
        return select(Llo, k)
    elif (k > below):
        return select(Lhi, k - below)
    else:
        return value

test = [6, 3, 2, 8, 4, 5, 1, 7, 0, 9]
print(select(test, 5))
```

- How fast?
- Is it really any better than sorting, and selecting?
Select with Good Splits

• Runtime depends on our selection of $m$:
  - A good selection will split $L$ evenly such that
    
    \[ |L_{lo}| = |L_{hi}| = \frac{|L|}{2} \]
  - The recurrence relation is:
    \[ T(n) = T(n/2) \]
    
    \[ n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \ldots = 2n \subseteq O(n) \]

Same as search for minimum
Select with Bad Splits

However, a poor selection will split L unevenly and in the worst case, all elements will be greater or less than \( m \) so that one Sublist is full and the other is empty.

For a poor selection, the recurrence relation is

\[
T(n) = T(n-1)
\]

In this case, the runtime is \( O(n^2) \).

Our dilemma:
\( O(n) \) or \( O(n^2) \),
depending on the list... or \( O(n \log n) \) independent of it.
Select Analysis (cont’d)

• Select seems risky compared to Sort
• To improve Select, we need to choose $m$ to give good ‘splits’
• It can be proven that to achieve $O(n)$ running time, we don’t need a perfect splits, just reasonably good ones.
• In fact, if both subarrays are at least of size $n/4$, then running time will be $O(n)$.
• This implies that half of the choices of $m$ make good splitters.
A Randomized Approach

- To improve Select, *randomly* select $m$.
- Since half of the elements will be good splitters, if we choose $m$ at random we will get a 50% chance that $m$ will be a good choice.
- This approach will make sure that no matter what input is received, the expected running time is small.
Randomized Select

```
In [48]:

import random

def randomizedSelect(L, k):
    value = random.choice(L)
    Llo = [t for t in L if t < value]
    Lhi = [t for t in L if t > value]
    below = len(Llo) + 1
    if (len(Llo) >= k):
        return randomizedSelect(Llo, k)
    elif (k > below):
        return randomizedSelect(Lhi, k - below)
    else:
        return value

test = [6, 3, 2, 8, 4, 5, 1, 7, 0, 9]
print(randomizedSelect(test, 5))
```

4
RandomizedSelect Analysis

• Worst case runtime: $O(n^2)$
• *Expected runtime*: $O(n)$.
• Expected runtime is a good measure of the performance of randomized algorithms, often more informative than worst case runtimes.
• Worst case runtimes are rarely repeated
• RandomizedSelect always returns the correct answer, which offers a way to classify Randomized Algorithms.
Types of Randomized Algorithms

- **Las Vegas Algorithms** – always produce the correct solution (i.e. randomizedSelect)

- **Monte Carlo Algorithms** – do not always return the correct solution.

Of course, Las Vegas Algorithms are always preferred, but they are often hard to come by.
Gibbs Sampling

• RandomProfileMotifSearch is probably not the best way to find motifs. Depends on random guesses followed by a greedy optimization procedure.

• Gibbs Sampling estimates a distribution of each variable in turn, conditional on the current values of the other variables.

• However, we can improve the algorithm by introducing Gibbs Sampling, an iterative procedure that discards one $k$-mer’s contribution to the profile distribution at each iteration and replaces it with a new one.

• Gibbs Sampling starts out slowly but chooses new $k$-mers with increasing the odds that it will improve the current solution.
How Gibbs Sampling Works

1) Randomly choose starting positions 
   \( s = (s_1, \ldots, s_t) \) and form the set of \( k \)-mers associated 
   with these starting positions.

2) Randomly choose one of the \( t \) sequences.

3) Create a profile \( P \) from the other \( t - 1 \) sequences.

4) For each position in the removed sequence, 
   calculate the probability that the \( l \)-mer starting at 
   that position was generated by \( P \).

5) Choose a new starting position for the removed 
   sequence at random based on the probabilities 
   calculated in step 4.

6) Repeat steps 2-5 until there is no improvement.
Gibbs Sampling: an Example

Input:

\[ t = 5 \text{ sequences, motif length } l = 8 \]

1. GTAAACAATATTTATAGC
2. AAAATTTACCTCGCAAGG
3. CCGTACTGTCAAGCGTGG
4. TGAGTAAACGACGTCCCA
5. TACTTAACACCCTGTCAA
Gibbs Sampling: an Example

1) Randomly choose starting positions, \( s=(s_1,s_2,s_3,s_4,s_5) \) in the 5 sequences:

\[
\begin{align*}
  s_1 & = 7 & \text{GTAAAC} & \text{GTAAAC}\ldots & \text{GTAAAC} \\
  s_2 & = 11 & \text{AATATTTT} & \text{AATATTTT}\ldots & \text{AATATTTT} \\
  s_3 & = 9 & \text{TTAGAAGG} & \text{TTAGAAGG}\ldots & \text{TTAGAAGG} \\
  s_4 & = 4 & \text{CCGTACTG} & \text{CCGTACTG}\ldots & \text{CCGTACTG} \\
  s_5 & = 1 & \text{ATTATTTT} & \text{ATTATTTT}\ldots & \text{ATTATTTT} \\
\end{align*}
\]
Gibbs Sampling: an Example

2) Choose one of the sequences at random:

**Sequence 2:** AAAATTTACCCTTTAGAAGG

\[
\begin{align*}
  s_1 &= 7 \quad \text{GTAAACAATATTTATAGC} \\
  s_2 &= 11 \quad \text{AAAATTTACCCTTTAGAAGG} \\
  s_3 &= 9 \quad \text{CCGTACTGTCAAGCGTGG} \\
  s_4 &= 4 \quad \text{TGAGTAAACGACGTCCCA} \\
  s_5 &= 1 \quad \text{TACTTTAACACCCTGTCAA}
\end{align*}
\]
Gibbs Sampling: an Example

2) Choose one of the sequences at random:

**Sequence 2:** AAAATTTACCTTAGAAGG

\[ s_1 = 7 \quad \text{GTAAAC}\text{AATATT}T\text{TATAGC} \]

\[ s_3 = 9 \quad \text{CCGTACTG}\text{TCAAGCGT}\text{GG} \]

\[ s_4 = 4 \quad \text{TGA}\text{GTAACCGA}\text{CGTCCCCA} \]

\[ s_5 = 1 \quad \text{TACTTTAAC}\text{ACCCTGTCAAA} \]
3) Create profile $P$ from $l$-mers in remaining 4 sequences:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>A</th>
<th>T</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>A</td>
<td>T</td>
<td>A</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>A</td>
<td>G</td>
<td>C</td>
<td>G</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>G</td>
<td>T</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>G</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>A</td>
<td>C</td>
<td>T</td>
<td>T</td>
<td>A</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1/4</td>
<td>2/4</td>
<td>2/4</td>
<td>3/4</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>2/4</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1/4</td>
<td>1/4</td>
<td>0</td>
<td>0</td>
<td>2/4</td>
<td>0</td>
<td>1/4</td>
</tr>
<tr>
<td>T</td>
<td>2/4</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>2/4</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
</tr>
<tr>
<td>G</td>
<td>1/4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1/4</td>
<td>0</td>
<td>3/4</td>
<td>0</td>
</tr>
<tr>
<td>Consensus String</td>
<td>T</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>G</td>
<td>A</td>
</tr>
</tbody>
</table>

Consensus String: T A A A T C G A
4) Calculate the $\text{prob}(a | P)$ for every possible 8-mer in the removed sequence:

Strings Highlighted in Red

| String | $\text{prob}(a | P)$ |
|--------|---------------------|
| AAAATTTACCTTTAGAAGG | .000732 |
| AAAATTTACCTTTAGAAGG | .000122 |
| AAAATTTACCTTTAGAAGG | 0 |
| AAAATTTACCTTTAGAAGG | 0 |
| AAAATTTACCTTTAGAAGG | 0 |
| AAAATTTACCTTTAGAAGG | 0 |
| AAAATTTACCTTTAGAAGG | 0 |
| AAAATTTACCTTTAGAAGG | .000183 |
| AAAATTTACCTTTAGAAGG | 0 |
| AAAATTTACCTTTAGAAGG | 0 |
| AAAATTTACCTTTAGAAGG | 0 |
5) Create a distribution of probabilities of $k$-mers $\text{prob}(a | P)$, and randomly select a new starting position based on this distribution.

A) To create this distribution, divide each probability $\text{prob}(a | P)$ by the total:

Starting Position 1: $\text{prob}(\text{AAAATTTA} | P) = .706$
Starting Position 2: $\text{prob}(\text{AAATTTAC} | P) = .118$
Starting Position 8: $\text{prob}(\text{ACCTTAGA} | P) = .176$
B) Select a new starting position at random according to computed distribution:

- \( P(\text{selecting starting position 1}): \, .706 \)
- \( P(\text{selecting starting position 2}): \, .118 \)
- \( P(\text{selecting starting position 8}): \, .176 \)

```python
t = random.random()
if (t < .706):
    # use position 1
elif (t < (.706 + .118)):
    # use position 2
else:
    # use position 8
```
Gibbs Sampling: an Example

Assume we select the substring with the highest probability – then we are left with the following new substrings and starting positions.

\[ s_1 = 7 \quad \text{GTAAAC} \text{AAT} \text{ATTTTATAGC} \]
\[ s_2 = 1 \quad \text{AAAATTTA} \text{CCTCGCAAGG} \]
\[ s_3 = 9 \quad \text{CCGTACTG} \text{TCAAGCGTGG} \]
\[ s_4 = 5 \quad \text{TGAGTAA} \text{TCGACGTCCCA} \]
\[ s_5 = 1 \quad \text{TACTTCAC} \text{ACCCTGTCAAA} \]
Gibbs Sampling: an Example

6) We iterate the procedure again with the above starting positions until we cannot improve the score any more.

```python
import numpy

def Score(seq, i, k, distr):
    return numpy.prod([distr[j][seq[i+j]] for j in range(k)])

def Profile(DNA, offset, k):
    profile = []
    t = len(DNA)
    for i in range(k):
        counts = {base : 0.01 for base in "acgt"}
        for j in xrange(t):  
            counts[DNA[j][offset[j]+i]] += 0.96 / t
        profile.append(counts)
    return profile
```
def GibbsProfileMotifSearch(seqList, k):
    start = [random.randint(0, len(seqList[t]) - k) for t in range(len(seqList))]
    bestScore = 0.0
    noImprovement = 0
    while True:
        remove = random.randint(0, len(seqList) - 1)
        start[remove] = -1
        distr = Profile(seqList, k, start)
        score = 0.0
        for t in range(len(seqList)):
            if (start[t] < 0):
                rScore = 0.0
                for i in xrange(len(seqList[remove]) - k + 1):
                    score = Score(seqList[remove], i, k, distr)
                    if (score > rScore):
                        rStart, rScore = i, score
                        score += rScore
                        start[t] = rStart
                else:
                    score += Score(seqList[t], start[t], k, distr)
            if (score > bestScore):
                bestScore = score
                noImprovement = 0
            else:
                noImprovement += 1
            if (noImprovement > len(seqList)):
                break
        return score, start
In [116]:

```python
random.seed(2020)

seqApprox = [
    'tagtggtctttttgagtctagatctgtaaggggaagaattttcaccagttccgggttcacccagcacagggtgacttaat',
    'cgccgacctggcgcctcagttatcgacgcttagacaaaccgggtttgatgccacgaaactttgagtttaatcggagttcct',
    'gtaactttgtgacgcctggttagaccggacaaatatcattgtggctgtcatagcggagctgacatcagataggggaaaatgcgt',
    'aacatcagggctttgtattaaacaatatttaacgacgttaatactccagttgacctgtagcacaatcgggaactgccggctccggg',
    'accacccgatataggtccttattaggttcaccgaaaaggtagtctcgtatcgaataataatggtctagccatgctcaatgtgggccttccac',
    'tagatcggatctcggatcgttctccttcccctcttggttaacgaggggtcgcacctttgtctgcatgtgccgaactttgtacc',
    'gaaatgggtcctgttgctgcatatcagccgcttttcccttatcccttgccgttgtagagtcatattccgttgtgctgcgct',
    'atgtatctagatcctatctccagctctctgttattgcccggagaacccttttctccacactacaagggctacttgtagctgcgct',
    'tttctacaccctctctttctgatcctcaaaacccgtttgggcccactttttctgctgctctgttacactctccatccttctgatgac',
    'ctactctatgtaaaaccaatcatcttaaccgtagctgccgcttttctgtatccctactacaggtcgagctccggaacttccg'
]

s, m = GibbsProfileMotifSearch(seqApprox, 10)

print(s, m)
for i, j in enumerate(m):
    print(seqApprox[i][j:j+10])
```

```output
0.0137569615302 [17, 47, 18, 33, 21, 0, 46, 70, 16, 65]
tagatctgaa
tgtatccgaa
tagacccgaa
taatccgaa	tagttccaa
tagatccgaa
cagatccgaa
tagatccgta
tagatccaa	tcgatccgaa
```
Gibbs Sampler in Practice

• Fewer profile searches, $O(n)$, in exchange for updating the profile, $O(kt)$, more often (tradeoff which is easier)
• Gibbs sampling can converge much faster than a fully randomized approach
• Gibbs sampling is more likely to converge to locally optimal motifs rather a fully randomized algorithm.
• Like the fully Randomized Algorithm it must be run with many randomly chosen initial seeds to achieve good results.
It’s Over

• Final Friday, 5/1
  – 8:00-11:00am
  – Be sure to sign into zoom
  – Open book, open notes, open internet, online
  – Will cover material since midterm
  – Final Study session:
    • Monday 4/27, 4pm-6pm