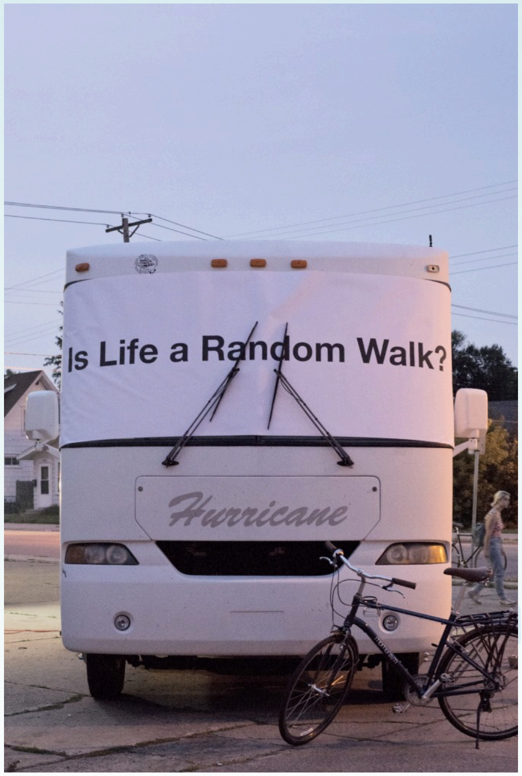
Randomized Algorithms



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

```
minv = HUGE
for v in L:
    if (v < minv):
        minv = v
```

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

Select Recursion

- **Select(L, k)** finds the kth smallest element in **L**
  - Select an element m from unsorted list **L** and partition L the array into two smaller lists:

$\mathbf{L}_{lo}$ - elements smaller than m

and

$\mathbf{L}_{hi}$ - elements larger than m

- If len($\mathbf{L}_{lo}$) > k then
  Select($\mathbf{L}_{lo}$, k)
  else if k > len($\mathbf{L}_{lo}$) + 1 then
  Select($\mathbf{L}_{hi}$, k – (len($\mathbf{L}_{lo}$) + 1))
  else m is the kth smallest element

Example of Select(L, 5)

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

<u>Step 1</u>: Choose the first element as *m*

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

Our Selection

Example of Select(L,5) (cont'd)

Step 2: Split the array into $\mathbf{L}_{lo}$ and $\mathbf{L}_{hi}$

$\mathbf{L}_{lo}$ = { 3, 2, 4, 5, 1, 0 }

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

$\mathbf{L}_{hi}$ = { 8, 7, 9 }

Example of Select(L,5) (cont'd)

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

len(L_{lo}) > k = 5 → Select({ 3, 2, 4, 5, 1, 0 }, 5)

m = 3

L_{lo} = { 2, 1, 0 } L_{hi} = { 4, 5 }

k = 5 > len(L_{lo}) +1 → Select({4, 5 }, 5 - 3 - 1)

m = 4

L_{lo} = { empty }, L_{hi} = { 5 }

k = 1 == len(L_{lo}) + 1 → return 4

Select Code

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

Select with Good Splits

- Runtime depends on our selection of m:

  - A good selection will split **L** evenly such that

$$|\mathbf{L}_{lo}| = |\mathbf{L}_{hi}| = |\mathbf{L}|/2$$

  - The recurrence relation is:

$$T(n) = T(n/2)$$

$$n + n/2 + n/4 + n/8 + n/16 + \ldots = 2n \rightarrow O(n)$$

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 $O(n^2)$.

I could have sorted first and done better

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

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

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.

Recall the Motif Finding Problem

Motif Finding Problem: Given a list of *t* sequences each of length *n*, find the "best" pattern of length *k* that appears in each of the *t* sequences.

$k = 8$

DNA

$t=5$

```
cctgatagacgctatctggctatccaGgtacTtaggtcctctgtgcgaatctatgcgtttccaaccat
agtactggtgtacatttgatCcAtacgtacaccggcaacctgaaacaaacgctcagaaccagaagtgc
aaacgtTAgtgcaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaatttt
agcctccgatgtaagtcatagctgtaactattacctgccacccctattacatcttacgtCcAtataca
ctgttatacaacgcgtcatggcggggtatgcgttttggtcgtcgtacgctcgatcgttaCcgtacgGc
```

$n = 69$

A New Motif Finding Approach

- **Motif Finding Problem**: Given a list of *t* length *n* sequences, find the best near-matching pattern of length *k* in each sequence.
- **Previously:** we have solved the Motif Finding Problem using a Branch-and-Bound or a Exhaustive techniques.
- **Now**: **Randomly** select possible locations and find a way to change those locations in an attempt to converge to the hidden motif.

Profiles Revisited

- Let **s** = (s_1,…,s_t) be the starting positions for *k*-mers in our *t* sequences.
- The substrings corresponding to these starting positions will form:

  - *t* x *k* ***alignment matrix***

  - 4 x *k* ***profile matrix****

* Note that we now define the profile matrix in terms of frequency, not counts as before.

l

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

t

A	0.6	0.0	0.2	0.0	0.6	0.2	0.2	0.0
C	0.4	0.8	0.0	0.0	0.2	0.8	0.0	0.0
G	0.0	0.2	0.8	0.0	0.0	0.0	0.6	0.2
T	0.0	0.0	0.0	1.0	0.2	0.0	0.2	0.8

4

x	a	c	g	t	a	c	g	t

P(X|profile)=0.6*0.8*0.8*1.0*0.6*0.8*0.6*0.8 = 0.0885

Scoring Strings with a Profile

- Let k-mer **a** = $a_1, a_2, a_3, \ldots a_k$
- $P(\mathbf{a}\,|\,\mathbf{P})$ is defined as the probability that an k-mer **a** was created by the Profile distribution **P**.
- If **a** is very similar to the consensus string of **P** then $P(\mathbf{a}\,|\,\mathbf{P})$ will be high
- If **a** is very different, then $P(\mathbf{a}\,|\,\mathbf{P})$ will be low.

$$Prob(\mathbf{a}\,|\,\mathbf{P}) = \prod_{i=1}^{k} p(a_i, i)$$

Scoring Strings with a Profile (cont'd)

Given a profile: **P** =

A	1/2	7/8	3/8	0	1/8	0
C	1/8	0	1/2	5/8	3/8	0
T	1/8	1/8	0	0	1/4	7/8
G	1/4	0	1/8	3/8	1/4	1/8

The probability of the consensus string:

Prob(**aaacct**|**P**) = ???

Scoring Strings with a Profile (cont'd)

Given a profile: **P** =

A	**1/2**	**7/8**	**3/8**	0	1/8	0
C	1/8	0	1/2	**5/8**	**3/8**	0
T	1/8	1/8	0	0	1/4	**7/8**
G	1/4	0	1/8	3/8	1/4	1/8

The probability of the consensus string:

Prob(**aaacct**|**P**) = 1/2 x 7/8 x 3/8 x 5/8 x 3/8 x 7/8 = .033646

Scoring Strings with a Profile (cont'd)

Given a profile: **P** =

A	**1/2**	7/8	**3/8**	0	**1/8**	0
C	1/8	0	1/2	**5/8**	3/8	0
T	1/8	**1/8**	0	0	1/4	7/8
G	1/4	0	1/8	3/8	1/4	**1/8**

The probability of the consensus string:

Prob(**aaacct**|**P**) = 1/2 x 7/8 x 3/8 x 5/8 x 3/8 x 7/8 = .033646

Probability of a different string:

Prob(**atacag**|**P**) = 1/2 x 1/8 x 3/8 x 5/8 x 1/8 x 1/8 = .001602

P-Most Probable *k*-mer

- Define the **P**-most probable *k*-mer from a sequence as an *k*-mer in that sequence which has the highest probability of being created from the profile **P**.

P =

A	1/2	7/8	3/8	0	1/8	0
C	1/8	0	1/2	5/8	3/8	0
T	1/8	1/8	0	0	1/4	7/8
G	1/4	0	1/8	3/8	1/4	1/8

Given a sequence = ctataaaccttacatc, find the *k*-mer that best matches the given profile

P-Most Probable *k*-mer (cont'd)

A	1/2	7/8	3/8	0	1/8	0
C	1/8	0	1/2	5/8	3/8	0
T	1/8	1/8	0	0	1/4	7/8
G	1/4	0	1/8	3/8	1/4	1/8

Find the *Prob*(**a**|**P**) of every possible 6-mer:

First try: **c t a t a a a c c t t a c a t c**

Second try: **c t a t a a a c c t t a c a t c**

Third try: **c t a t a a a c c t t a c a t c**

-Continue this process to evaluate every possible 6-mer

P-Most Probable *k*-mer (cont'd)

Compute *prob*(**a**|**P**) for every possible 6-mer:

String, Highlighted in Red	Calculations	*prob*(**a** | **P**)
ctataaaccttacat	1/8 x 1/8 x 3/8 x 0 x 1/8 x 0	0
ctataaaccttacat	1/2 x 7/8 x 0 x 0 x 1/8 x 0	0
ctataaaccttacat	1/2 x 1/8 x 3/8 x 0 x 1/8 x 0	0
ctataaaccttacat	1/8 x 7/8 x 3/8 x 0 x 3/8 x 0	0
ctataaaccttacat	1/2 x 7/8 x 3/8 x 5/8 x 3/8 x 7/8	.0336
ctataaaccttacat	1/2 x 7/8 x 1/2 x 5/8 x 1/4 x 7/8	.0299
ctataaaccttacat	1/2 x 0 x 1/2 x 0 1/4 x 0	0
ctataaaccttacat	1/8 x 0 x 0 x 0 x 0 x 1/8 x 0	0
ctataaaccttacat	1/8 x 1/8 x 0 x 0 x 3/8 x 0	0
ctataaaccttacat	1/8 x 1/8 x 3/8 x 5/8 x 1/8 x 7/8	.0004

P-Most Probable *k*-mer (cont'd)

P-Most Probable 6-mer in the sequence is aaacct:

String, Highlighted in Red	**Calculations**	*Prob*(**a** | **P**)
ctataaaccttacat	1/8 x 1/8 x 3/8 x 0 x 1/8 x 0	0
ctataaaccttacat	1/2 x 7/8 x 0 x 0 x 1/8 x 0	0
ctataaaccttacat	1/2 x 1/8 x 3/8 x 0 x 1/8 x 0	0
ctataaaccttacat	1/8 x 7/8 x 3/8 x 0 x 3/8 x 0	0
ctataaaccttacat	**1/2 x 7/8 x 3/8 x 5/8 x 3/8 x 7/8**	**.0336**
ctataaaccttacat	1/2 x 7/8 x 1/2 x 5/8 x 1/4 x 7/8	.0299
ctataaaccttacat	1/2 x 0 x 1/2 x 0 1/4 x 0	0
ctataaaccttacat	1/8 x 0 x 0 x 0 x 0 x 1/8 x 0	0
ctataaaccttacat	1/8 x 1/8 x 0 x 0 x 3/8 x 0	0
ctataaaccttacat	1/8 x 1/8 x 3/8 x 5/8 x 1/8 x 7/8	.0004

P-Most Probable *k*-mer (cont'd)

aaacct is the **P**-most probable 6-mer in:

ctataaaccttacatc

because *Prob*(**aaacct**|**P**) = .0336 is greater than the *Prob*(**a**|**P**) of any other 6-mer in the sequence.

Dealing with Zeroes

- In our toy example *prob*(**a** | **P**)=0 in many cases. In practice, there will be enough sequences so that the number of elements in the profile with a frequency of zero is small.
- To avoid many entries with *prob*(**a** | **P**)=0, there exist techniques to equate zero to a very small number so that one zero does not make the entire probability of a string zero. Pseudo counts (assigning a *prior* probability based on our best guess).

P-Most Probable *k*-mers in Many Sequences

- Find the **P**-most probable *k*-mer in each of the "t" sequences.

P=

A	1/2	7/8	3/8	0	1/8	0
C	1/8	0	1/2	5/8	3/8	0
T	1/8	1/8	0	0	1/4	7/8
G	1/4	0	1/8	3/8	1/4	1/8

```
ctataaacgttacatc
atagcgattcgactg
cagcccagaaccct
cggtataccttacatc
tgcattcaatagctta
tatcctttccactcac
ctccaaatcctttaca
ggtcatcctttatcct
```

P-Most Probable *k*-mers in Many Sequences (cont'd)

1	a	a	a	c	g	t
2	a	t	a	g	c	g
3	a	a	c	c	c	t
4	g	a	a	c	c	t
5	a	t	a	g	c	t
6	g	a	c	c	t	g
7	a	t	c	c	t	t
8	t	a	c	c	t	t
A	5/8	5/8	4/8	0	0	0
C	0	0	4/8	6/8	4/8	0
T	1/8	3/8	0	0	3/8	6/8
G	2/8	0	0	2/8	1/8	2/8

ctataaacgttacatc

atagcgattcgactg

cagcccagaaccct

cggtgaaccttacatc

tgcattcaatagctta

tgtcctgtccactcac

ctccaaatcctttaca

ggtctacctttatcct

P-Most Probable *k*-mers give a new profile

Comparing New and Old Profiles

1	a	a	a	c	g	t
2	a	t	a	g	c	g
3	a	a	c	c	c	t
4	g	a	a	c	c	t
5	a	t	a	g	c	t
6	g	a	c	c	t	g
7	a	t	c	c	t	t
8	t	a	c	c	t	t
A	5/8	5/8	4/8	0	0	0
C	0	0	4/8	6/8	4/8	0
T	1/8	3/8	0	0	3/8	6/8
G	2/8	0	0	2/8	1/8	2/8

A	**1/2**	**7/8**	**3/8**	**0**	**1/8**	**0**
C	**1/8**	**0**	**1/2**	**5/8**	**3/8**	**0**
T	**1/8**	**1/8**	**0**	**0**	**1/4**	**7/8**
G	**1/4**	**0**	**1/8**	**3/8**	**1/4**	**1/8**

Red – frequency increased, Blue – frequency decreased

Random Profile Motif Search

Use P-Most probable *k*-mers to adjust start positions until we reach a "best" profile; this is the motif.

1) Select random starting positions.
3) Create a profile **P** from the substrings at these starting positions.
4) Find the **P**-most probable *k*-mer **a** in each sequence and change the starting position to the starting position of **a**.
5) Compute a new profile based on the new starting positions after each iteration and proceed until we cannot increase the score anymore.
6) Repeat the entire process (Steps 1-5) a few times and keep the best answer.

RandomProfileMotifSearch Algorithm

```
def Profile(seqList, k, start):
    dist = [dict([(base,0.1) for base in "acgt"]) for i in xrange(k)]
    # Count base occurrences in each column
    for t in xrange(len(seqList)):
        for i, base in enumerate(seqList[t][start[t]:start[t]+k]):
            dist[i][base] += 1.0
    # Normalize (divide by total)
    for i in xrange(k):
        total = sum(dist[i].values())
        for base in "acgt":
            dist[i][base] /= total
    # return Distribution
    return dist

def Score(seq, si, k, dist):
    prob = 1.0
    for i, base in enumerate(seq[si:si+k]):
        prob *= dist[i][base]
    return prob
```

RandomProfileMotifSearch Algorithm

```
def RandomProfileMotifSearch(seqList, k):
    start = [random.randint(0,len(seqList[t])-k+1) for t in xrange(len(seqList))]
    bestScore = 0.0
    while True:
        distr = Profile(seqList, k, start)
        score = 0.0
        for t in xrange(len(seqList)):
            score += Score(seqList[t], start[t], k, distr)
        if (score <= bestScore):
            break
        bestScore = score
        for t in xrange(len(seqList)):
            newStart, newScore = -1, 0.0
            for i in xrange(len(seqList[t])-k+1):
                score = Score(seqList[t], i, k, distr)
                if (score > newScore):
                    newStart = i
                    newScore = score
            start[t] = newStart
    return score, start
```

Example

```
def FindMotif(seqList, k, N):
    highScore = 0.0
    for i in xrange(N):
        score, start = RandomProfileMotifSearch(seqList, k)
        if score > highScore:
            motif = [s for s in start]
            highScore = score
    return highScore, motif

%timeit s, m = FindMotif(seqApprox, 10, 100)
print s
for i, si in enumerate(m):
    print si, seqApprox[i][si:si+10]
```

```
1 loops, best of 3: 457 ms per loop
0.297843115489
17 tagatctgaa
47 tggatccgaa
18 tagacccgaa
33 taaatccgaa
21 taggtccaaa
0 tagattcgaa
46 cagatccgaa
70 tagatccgta
16 tagatccaaa
65 tcgatccgaa
```

RandomProfileMotifSearch Analysis

- Since we choose starting positions randomly, there is little chance that our guess will be close to an optimal motif, meaning it will take a very long time to find the optimal motif.
- It is unlikely that the random starting positions will lead us to the correct solution at all.
- In practice, this algorithm is run many times, $O(n)$, with the hope that random starting positions will be close to the optimum solution simply by chance.
- Can we do better than a random guess and then following a greedy path?