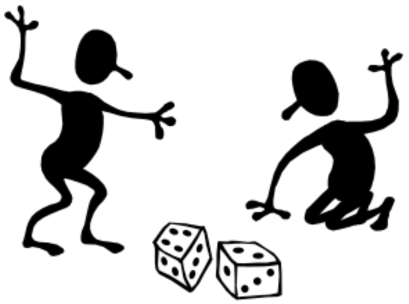
Randomized Algorithms

- Problem Set #4 has been graded
- Problem Set #5 by Thursday

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 Algorithm

- ***Select(L, k)*** finds the k^{th} smallest element in L
- *Select(L,1)* find the smallest element in a list:
  - Well known $O(n)$ algorithm

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

- *Select(L, len(L)/2)* finds the median...
  - How?
  - median = sorted(L)[len(L)/2] → $O(nlogn)$
- Can we find medians, or 1st quartiles in O(n)?

Recursive Select

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

    L_{lo} - elements smaller than m
    L_{hi} - elements larger than m
    if len(L_{lo}) > k:
        Select(L_{lo}, k)
    elif k > len(L_{lo}) + 1:
        Select(L_{hi}, k - len(L_{lo}) - 1)
    else:
        m is the k^{th} smallest element

Example of *Select(L,5)*

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

- **Step 1:** Choose the first element as m

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

- **Step 2:** Split into two lists

$$L_{lo} = \{{\color{blue}3, 2, 4, 5, 1, 0}\}$$
$$L_{hi} = \{{\color{green}8, 7, 9}\}$$

Example of *Select(L,5)* (continued)

- **Step 3** Recurively call Select on either L_{lo} or L_{hi} until len(L_{lo}) = k, then return *m*.

$$
\begin{aligned}
& len(L_{lo}) > k = 5 \rightarrow \\
& \qquad Select(\{3, 2, 4, 5, 1, 0\}, 5) \\
& \qquad m = 3 \qquad L_{lo} = \{2, 1, 0\} \qquad L_{hi} = \{4, 5\} \\
& k = 5 > len(L_{lo}) + 1 \rightarrow \\
& \qquad Select(\{4, 5\}, 5 - 3 - 1) \\
& \qquad m = 4 \qquad L_{lo} = \{\} \qquad L_{hi} = \{5\} \\
& k = 1 > len(L_{lo}) + 1 \rightarrow \\
& \qquad return\ 4
\end{aligned}
$$

Select in Python

```
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 (k < len(Llo)):
        return select(Llo, k)
    elif (k > below):
        return select(Lhi, k - below)
    else:
        return value

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

```
4
```

Select(L,k) Performance

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:

$$T(n) = T\left(\frac{n}{2}\right)$$

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \ldots = 2n \rightarrow O(n)$$

- which is the same as the seach for *min(L)* or *max(L)*

Select(L,k) 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 $O(n^2)$, which is worse than sorting first and selecting the k^{th} value

Our dimemma: $O(n)$ or $O(n^2)$ depending on the list... or $O(n \log n)$ independent of it

Select(L,k) verdict

- 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(L,k)*, 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

```
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 (k < len(Llo)):
        return randomizedSelect(Llo, k)
    elif (k > below):
        return randomizedSelect(Lhi, k-below)
    else:
        return value

print randomizedSelect([6,3,2,8,4,5,1,7,0,9], 5)
%timeit randomizedSelect(range(10000), 500)
```

```
5
100 loops, best of 3: 1.97 ms per loop
```

RandomizedSelect(L,k) Performance

- 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(L,k)* always returns the correct answer, which offers a way to classify Randomized Algorithms.

Two 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.
- Las Vegas Algorithms are always preferred, but they are often hard to come by.

The Motif Finding Problem

Given a list of t sequences each of length n, find the "best" matching pattern of length l that appears in each of the t sequences.

A New Approach

- ***Motif Finding Problem:*** Given a list of *t* sequences each of length *n*, find the “best” pattern of length *l* that appears in each of the *t* sequences.
- ***Previously:*** we solved the Motif Finding Problem using a Branch and Bound or a Greedy technique.
- ***Now:*** *Randomly* select possible locations and find a way to greedily change those locations until we converge to the hidden motif.

Profiles Revisited

- Let $s = (s_1, s_2, \ldots, s_t)$ be the starting positions for l-mers in our t sequences.
- The substrings corresponding to these starting positions will form:
  - $t \times l$ alignment matrix
  - $4 \times l$ profile matrix
- Normalized counts that they represent the fraction of each base at each position

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
x	a	c	g	t	a	c	g	t

4

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

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

Scoring with a Profile

Given the profile: $P =$

base	1	2	3	4	5	6
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 consequence string:
$Prob(aaacct|P) = 1/2 \times 7/8 \times 3/8 \times 5/8 \times 3/8 \times 7/8 = 0.033646$

The probability of a different string:
$Prob(atacag|P) = 1/2 \times 1/8 \times 3/8 \times 5/8 \times 1/8 \times 1/8 = 0.001602$

P-Most Probable k-mer

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

base	1	2	3	4	5	6
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 P-most probable k-mer

Find the $Prob(\boldsymbol{a}|P)$ of every possible 6-mer

```
CTATAAACCTTACATC
CTATAAACCTTACATC
CTATAAACCTTACATC
CTATAAACCTTACATC
CTATAAACCTTACATC
CTATAAACCTTACATC
CTATAAACCTTACATC
CTATAAACCTTACATC
```

P-Most Probable k-mer

Compute $Prob(\boldsymbol{a}|P)$ of every possible 6-mer

String highlighted in red	Path	Prob
CTATAAACCTTACATC	1/8×1/8×3/8×0×1/8×0	0
CTATAAACCTTACATC	1/2×7/8×0×0×1/8×0	0
CTATAAACCTTACATC	1/2×1/8×3/8×0×1/8×0	0
CTATAAACCTTACATC	1/8×7/8×3/8×0×3/8×0	0
CTATAAACCTTACATC	1/2×7/8×3/8×5/8×3/8×7/8	**0.0336**
CTATAAACCTTACATC	1/2×7/8×1/2×5/8×1/4×7/8	0.0299
CTATAAACCTTACATC	1/2×0×1/2×0×1/4×0	0
CTATAAACCTTACATC	1/8×0×0×0×1/8×0	0
CTATAAACCTTACATC	1/8×1/8×0×0×3/8×0	0
CTATAAACCTTACATC	1/8×1/8×3/8×5/8×1/8×7/8	0.0004
CTATAAACCTTACATC	1/8×7/8×1/2×0×1/4×0	0

- AAACCT is the P-most probable 6-mer

Dealing with Zeros

- In our toy example $Prob(\boldsymbol{a}|P)$ 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 $Prob(\boldsymbol{a}|P)$, 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 (assigning a *prior* probability, we will not address these techniques here).

P-Most Probable k-mers in Many Sequences

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

base	1	2	3	4	5	6
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
atagcgattcgactga
cagcccagaaccctgg
cggtgaaccttacatc
tgcattcaatagctta
tgtcctgtccactcac
ctccaaatcctttaca
ggtctacctttatcct
```

Next Time

- A consensus of consensus
- When does randomization show up?

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
atagcgattcgactga
cagcccagaaccctgg
cggtgaaccttacatc
tgcattcaatagctta
tgtcctgtccactcac
ctccaaatcctttaca
ggtctacctttatcct
```