Quiz Section Week 9
May 24, 2016

Hash-based alignment
Spaced seeding
What is hashing?

• A hash function maps some object $x$ to an integer $i$

• A hash function allows us to have a hash table, which is like a list that allows indexing by arbitrary objects (a python Dictionary!)

• We can compute the value of the hash function and find the index in the hash table in constant time

hash('hello') → 3

Hash table with key ‘hello’
Hash functions aren’t perfect

• There’s no practical function that can map every object in the universe to a unique integer

• Multiple keys can map to the same index in the hash table

• Hash table implementations have to somehow deal with “collisions”

hash(‘hello’) → 3
hash(‘goodbye’) → 3
hash(123.456) → 3

‘hello’
‘goodbye’
123.456
Hashing Improves Search

• A **hash function** assigns a unique key to each unique
data element (DNA sequence in our case)

\[
\text{hash(“ATGCTG”)} = \text{key1} \\
\text{hash(“TTTCTG”)} = \text{key2} \\
\ldots
\]

• **Keys** encode strings in a short, easily comparable
  format (e.g. a number)
Hashing Improves Search

- A **hash function** assigns a unique key to each unique data element (DNA sequence in our case)

- The **hash table** is an associative array that describes the relationship between the key and the sequence and its genomic location

<table>
<thead>
<tr>
<th>Key</th>
<th>Hashed index</th>
<th>Genomic location</th>
</tr>
</thead>
<tbody>
<tr>
<td>“GCTAGC”</td>
<td>Key1</td>
<td>Chr1 123412</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>“TTTAGC”</td>
<td>KeyN</td>
<td>Chr6 988472</td>
</tr>
</tbody>
</table>
The basic idea of hashing is to transform character strings into keys that can be easily compared (in fact, the key can point directly to a physical memory address)
Hashing Improves Search

A hashed reference genome

A read
CAGGTCGCA

Its key
0x4ad8
0x2b61
What next?
Hashing Improves Search

A hashed reference genome

A read: CAGGTCGCA
Its key: 0x4ad8
We can see that this key is in our hashed reference!
Create a hash table that maps all observed 4-mers to its position(s) in the reference genome ‘s’

\[
\begin{align*}
\text{s} &= \text{'ACAAGATGCCATTGTCCCCC CGGCCTCCTGCTGCTGCTGCTC T'} \\
\text{k} &= 4 \ # \ \text{size of the key} \\
\text{h} &= \{ \} \ # \text{'ACAA':[0], 'CCCC':[15,16]} \\
\end{align*}
\]
Create a hash table that maps all observed 4-mers to its position(s) in the reference genome ‘s’

s = 'ACAGATGCCCCATGGCTCCCCCAGGCTCTGCTGCTGCTGCTCT'
k = 4  # size of the key
h = {}  # 'ACAA':[0], 'CCCC':[15,16]
for i in range(0,len(reference)-k):
    s = reference[i:(i+k)]
    if s in h:
        h[s].append(i)
    else:
        h[s] = [i]
print h
s = 'ACAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCTCT'
print h

{
'CGGC': [19],  
'ACAA': [0],  
'GTCC': [13],  
'GGCC': [20],  
'AAGA': [2],  
'TTGT': [11],  
'ATTG': [10],  
'C CGG': [18],  
'AGAT': [3],  
'GATG': [4],  
'ATGC': [5],  
'GCTC': [37],  
'GCCA': [7],  
'CAAG': [1],  
'CCAT': [8],  
'CCCC': [15, 16],  
'TGCC': [6],  
'GCCT': [21],  
'CCCCG': [17],  
'TGCT': [27, 30, 33, 36],  
'CCTC': [22],  
'CCTG': [25],  
'TGTC': [12],  
'TCCT': [24],  
'CATT': [9],  
'GCTG': [28, 31, 34],  
'CTGC': [26, 29, 32, 35],  
'CTCC': [23],  
'TCCC': [14]}

Is this really faster than using .index()?

time.time() measures time!

import time

print time.time()  # Prints the number of seconds that have passed since January 1st, 1970
1464055997.75

start_time = time.time()
# Set of commands for which we want to measure running time
for i in range(0,1000000):
    do = 'nothing'
print time.time()-start_time  # Now print out running time
0.372000217438
Given a list of reads, find where in the reference genome they reside and print how long it takes

# Given h from before, fill the list
locations = []
# With the reads in list reads
reads = h.keys() * 1000
# And print how long it takes
Given a list of reads, find where in the reference genome they reside and print how long it takes

# Given h from before, fill the list
locations = []
# With the reads in list reads
reads = h.keys() * 1000
# And print how long it takes
start = time.time()
for s in reads:
    locations.append(h[s])
print 'dictionary:', time.time() - start
dictionary: 0.00799989700317
How does it compare to using `reference.index()`?

```python
# Using `.index()`, fill
locations = []
# With the reads in list `reads`
reads = h.keys() * 1000
# And print how long it takes
```
How does it compare to using reference.index()?

# Using .index(), fill
locations = []
# With the reads in list reads
reads = h.keys()*1000
# And print how long it takes
start = time.time()
for s in reads:
    locations.append( reference.index(s) )
print 'reference.index:', time.time()-start
.index: 0.0120000839233
Is this a fair comparison? What’s missing?
Is this a fair comparison? What’s missing?

```python
h = {}
k = 6
start = time.time()
for i in range(0, len(reference) - k):
    s = reference[i:(i+k)]
    if s in h:
        h[s].append(i)
    else:
        h[s] = [i]
print h
print 'constructing dictionary:', time.time() - start
constructing dictionary: 0.0440001487732
```
Is this a fair comparison? What’s missing?

constructing dictionary: 0.0440001487732
Using the dictionary: 0.00799989700317
Using reference.index: 0.0120000839233

Constructing the dictionary is expensive, but you only have to do it once, and you keep reaping the benefits
What if we want to do use hash alignment but don’t want to require exact matches?
Refinement #1: Spaced seeds

• Split each big seed into four “spaced” seeds
Refinement #1: Spaced seeds

- Split each seed into four “spaced” seeds
- Any read with two (or fewer) mismatches will have at least two seeds that match the reference (what do we mean here, exactly?)
Refinement #1: Spaced seeds

• Split each seed into four “spaced” seeds

• Any read with two (or fewer) mismatches will have **at least two seeds that match the reference**

• As usual, these hits can be further examined to determine how well they align
How, Exactly, Do We Do This?

How can we use hashing to search for any two spaced seed matches for a read?
How, Exactly, Do We Do This?

Rather than 1 hash table, store multiple hash tables that correspond to all the possible ways to get two mismatches amongst our four spaced seeds.
How, Exactly, Do We Do This?

• Rather than 1 hash table, store 6 hash tables
• Imagine we have 8 bp seed
• Create 6 hash tables (as below) where only positions with 1 are used (and 0 positions ignored).
• Even with mismatches at any 2 positions, 1+ hash tables will have a match.
• Follow up hits by counting total # of mismatches across seed. Extend alignment to full read. Report best hit for each read.

1: 11110000
2: 00001111
3: 11000011
4: 00111100
5: 11001100
6: 00110011

All possible pairs of positions (representing potential mismatch locations) are masked in at least 1 of the 6 hash tables
How, Exactly, Do We Do This?

Which sub-seed would successfully map this read to the genome?

1: 11110000
2: 00001111
3: 11000011
4: 00111100
5: 11001100
6: 00110011
How, Exactly, Do We Do This?

Sub-seed pair 6 would successfully map in hash table 6

1: 11110000
2: 00001111
3: 11000011
4: 00111100
5: 11001100
6: 00110011
How, Exactly, Do We Do This?

\[
s = 'ACAAGATGCCATTGCTCCCCCGGCCTCCTGCTGCTGCTGCTCT' \\
read = 'ACAAGATG' \\
mismatched_read = 'TCAAGTTG' \quad \# \text{Two mismatches}
\]

1: 11110000  TCAA----  Sub-seed 1 does not map to reference
2: 00001111
3: 11000011
4: 00111100
5: 11001100
6: 00110011
How, Exactly, Do We Do This?

\[
s = \text{'ACAAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCT'}
\]
\[
\text{read} = \text{'ACAAAGATG'}
\]
\[
\text{mismatched\_read} = \text{'TCAAGTTG'} \quad \# \text{Two mismatches}
\]

1: 11110000
2: 00001111
3: 11000011
4: 00111100
5: 11001100
6: 00110011  --AA--TG  \quad \text{Sub-seed 6 does map to reference}
# Given a seed 'sequence'
# Generate the string for sub-seed pair number $i$

```python
sequence = 'ACAAGATG'
def subseed( i, sequence ):
  pass
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11110000</td>
</tr>
<tr>
<td>1</td>
<td>00001111</td>
</tr>
<tr>
<td>2</td>
<td>11000011</td>
</tr>
<tr>
<td>3</td>
<td>00111100</td>
</tr>
<tr>
<td>4</td>
<td>11001100</td>
</tr>
<tr>
<td>5</td>
<td>00110011</td>
</tr>
</tbody>
</table>
How, Exactly, Do We Do This?

```python
def subseed( i, sequence ):
    if i==0:
        return sequence[:4]
    elif i == 1:
        return sequence[4:]
    elif i == 2:
        return sequence[:2] + sequence[6:]
    elif i == 3:
        return sequence[2:6]
    elif i == 4:
        return sequence[:2] + sequence[4:6]
    elif i == 5:
        return sequence[2:4] + sequence[6:]
```

0: 11110000
1: 00001111
2: 11000011
3: 00111100
4: 11001100
5: 00110011
How, Exactly, Do We Do This?

```python
sequence = 'ACAAGATG'
print subseed( 0, sequence )  # ACAA
print subseed( 1, sequence )  # GATG
print subseed( 2, sequence )  # ACTG
print subseed( 3, sequence )  # AAGA
print subseed( 4, sequence )  # ACGA
print subseed( 5, sequence )  # AATG
```
Construct and print 6 hash tables with the 6 subseeds given the subseed function from before

\[
s = \text{'ACAAGATGCCATTTGTCccccccccCGCCCTCCTGCTGCTGCTGCTC'}
\]

\[
k = 8 \quad \# \text{size of the key}
\]

\[
\text{subhashes} = \left[ \{ \} \text{ for } i \text{ in range}(0, 6) \right] \quad \# \text{initialize 6 sub hash tables}
\]

print subhashes
Construct 6 hash tables with the 6 subseeds

s = 'ACAAGATGCCATTGTCCCCCGCCTCCTGCTGCTGCTGCTCTCT'
k = 8 # size of the key
subhashes = [ {} for i in range(0,6) ] # initialize sub hash tables
for i in range(0,len(reference)-k): # i is position in reference
    s = reference[i:(i+k)] # seed
    for j in range(0,6): # iterate through subhashes
        sub_s = subseed(j,s) # jth subseed of s
        if sub_s in subhashes[j]:
            subhashes[j][sub_s].append(i)
        else:
            subhashes[j][sub_s] = [i]
print subhashes
Print location of a read of length 8 with 2 mismatches

\[ s = 'ACAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCT' \]
\[ \text{read} = 'ACAAGATG' \]
\[ \text{mismatched_read} = 'TCAAGATT' \]  # Two mismatches
Print location of a read of length 8 with 2 mismatches

s = 'ACAAAGATGCCATTGTCCCCCAGGGCCTCCTGCTGCTGCTGCTCT'
read = 'ACAAAGATG'
mismatched_read = 'TCAAGATT' # Two mismatches

for i in range(0,len(subhashes)):
    sub_s = subseed(i,mismatched_read)
    if sub_s in subhashes[i]:
        print 'Mismatched read location: ', subhashes[i][sub_s]

Mismatched read location: [0]