Quiz Section Week 9
May 23, 2017

A couple notes on homework, machine learning

High-throughput sequencing applications

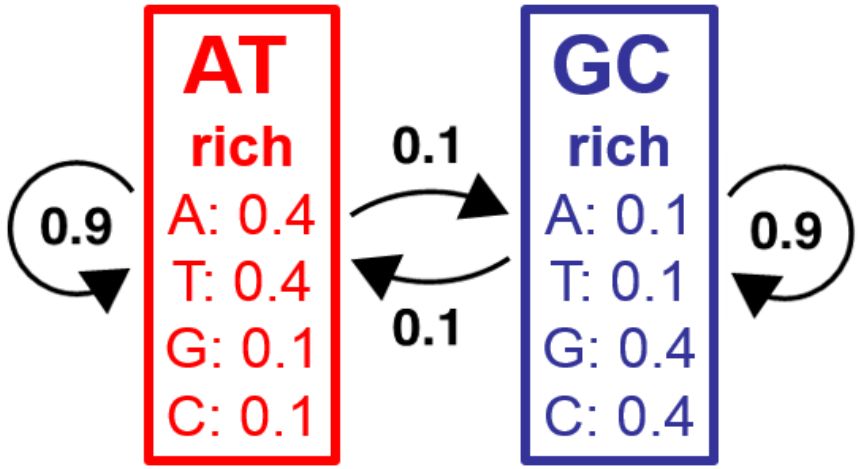
Hash-based alignment in Python

Quick recap: forward-backward

$$P(\pi_i = k|x) = \frac{P(x, \pi_i = k)}{P(x)}$$

$$P(x, \pi_i = k) = \sum_{\pi_{i=k}} P(\pi|x)$$

$$= f(i) * b(i)$$



	T	G	A
AT Rich (.5)	.5*.4 = .2	. .	. .1*.4 = .04
GC Rich (.5)	.5*.1 = .05	.2*.1*.4 = .008 .05*.9*.4 = .018	. 9*.1 = .09

Forward (RED): .008 + .018 = .026

Backward (GREEN): .04 + .09 = .13

The probability that the **GC rich state** emitted the **nucleotide G** in the observed sequence is .026*.13 = .00338

Reminder: Functions are most helpful when they are *modular* and *reusable*

```
import sys
import random

def read_transitions(input_file):
      transition_probs = {'A':{'A':{},
'T':{}}, 'T':{'A':{}, 'T':{}}}
      fin = open(input_file,'r')
      for line in fin:
            probs =
line.rstrip().split()
      transition_probs[probs[0]][probs
[1]] = float(probs[2])
      fin.close()
      return transition_probs
```

```
def markov_step(transition_probs,
current_state):
      trans_prob =
transition_probs[current_state]

      prob_choice =
random.random()
      if prob_choice <
trans_prob['A']:
            return 'A'
      else:
            return 'T'
```

Think of functions as tools you build to help yourself out

```
if __name__ == "__main__":
        input_file = sys.argv[1]
        input_file = sys.argv[1]
        trans_probs = read_transitions(input_file)
#       trans_probs = {'A':{'A':0.8, 'T':0.2}, 'T':{'A':0.2, 'T':0.8}}
        seq_len = int(sys.argv[2])
        start = random.random()
        if start < 0.5:
                seq = 'A'
        else:
                seq = 'T'
        for i in range(seq_len-1):
                seq = seq + markov_step(trans_probs, seq[i])
        print seq
```

Supervised machine learning: another way to think about it

- Given N training examples (objects)
  - $\{(x_1,y_1), (x_2,y_2),\ldots, (x_N,y_N)\}$
    Features and class labels
- Machine learning algorithm finds a function g: X → Y
  - Decision Tree
  - HMM
- Parameters of g are trained from an "objective function"
  - Decision tree: branch purity
  - Maximum likelihood

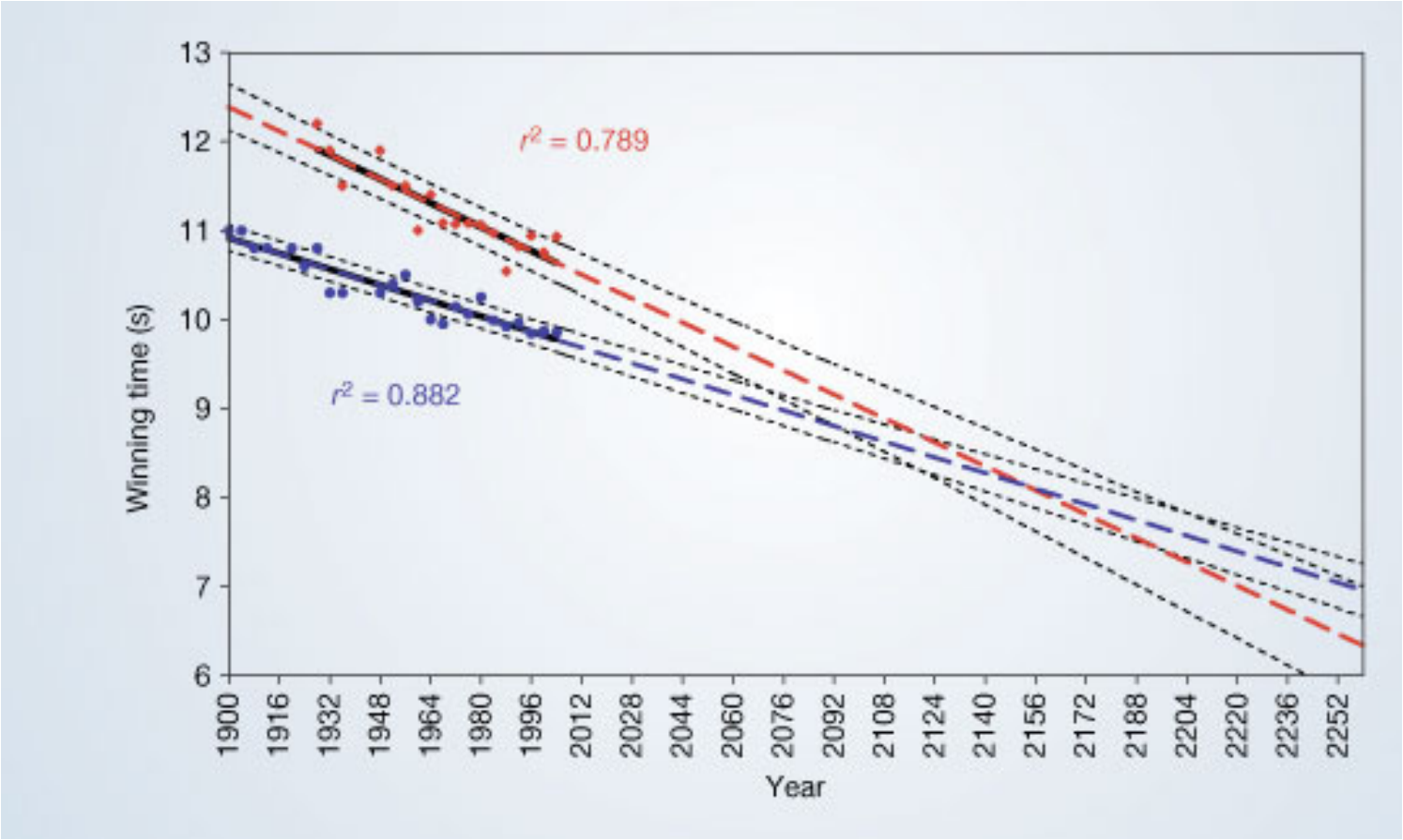
Mapping function g	features	Parameters θ	Optimization criterion
Decision Tree	Binary variables (but could a lot of things)	How/when to branch	Branch purity
HMM	Nucleotides (but could be anything)	emission and transition probabilities	P(x|θ)
SVM	Numbers	hyperplane weights	Maximum-margin
Linear Regression	?	?	?

Questions for evaluating machine learning models

(tl;dr: read the methods! be a skeptic!)

- How were features chosen? Are there data quality concerns?
  - Garbage in, garbage out
  - poorly designed experiments, biased data, batch effects
- Are we evaluating on training, validation, or test data? How were the datasets chosen? Any circularity?
  - Changing model choices based on held-out data
  - Variant effect predictors only trained on clearly benign or deleterious variants
- Are model assumptions valid?
- What are the limits of the training and testing data? How generalizable is this model?
  - Variant effect predictors only trained and tested on European genetic backgrounds
- Are samples balanced between positive and negative? How is this accounted for?
- What metrics were used for evaluation? What metrics are not shown?
  - http://sphweb.bumc.bu.edu/otlt/mph-modules/bs/bs704_probability/bs704_probability4.html

100m dash Olympic gold medal times
(Tatem *Nature* 2004)

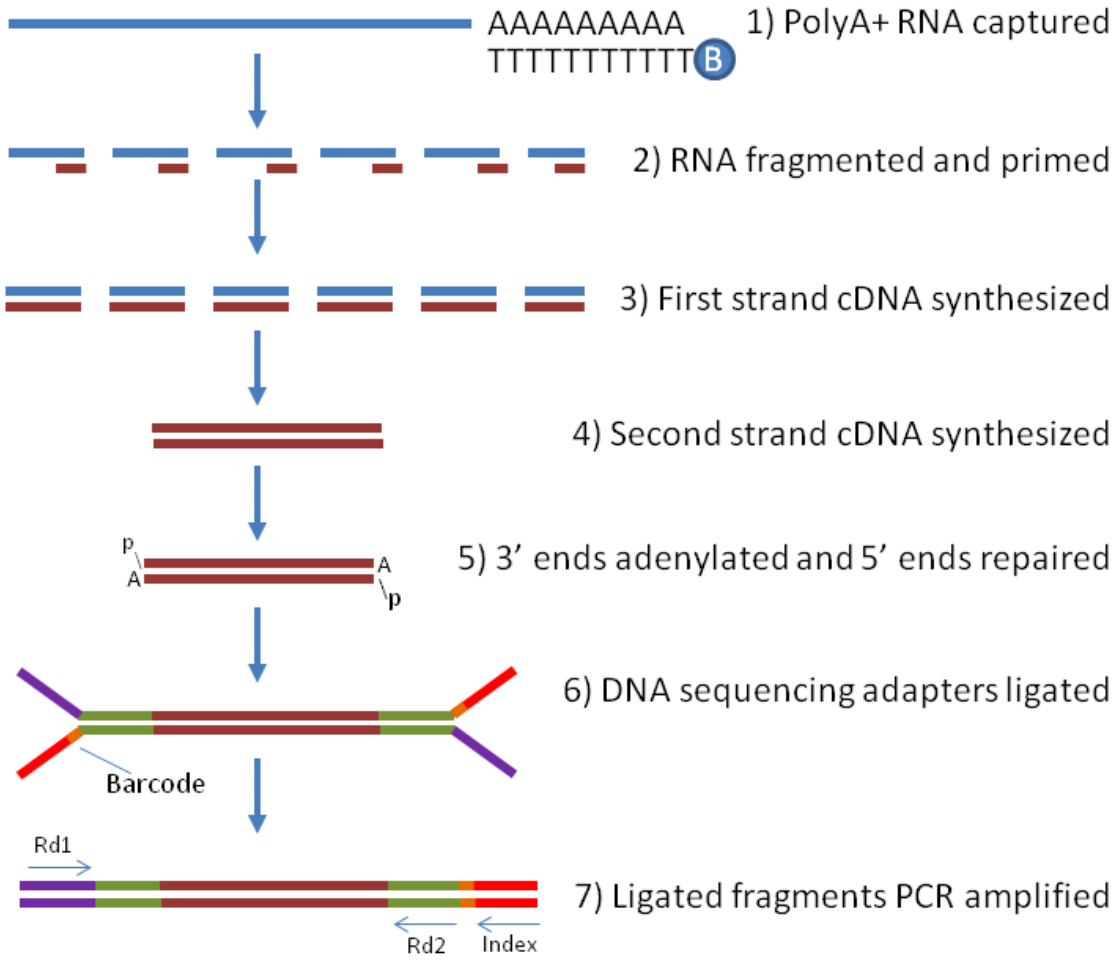


Questions about high-throughput sequencing?

Sequencing as tool for biological measurement

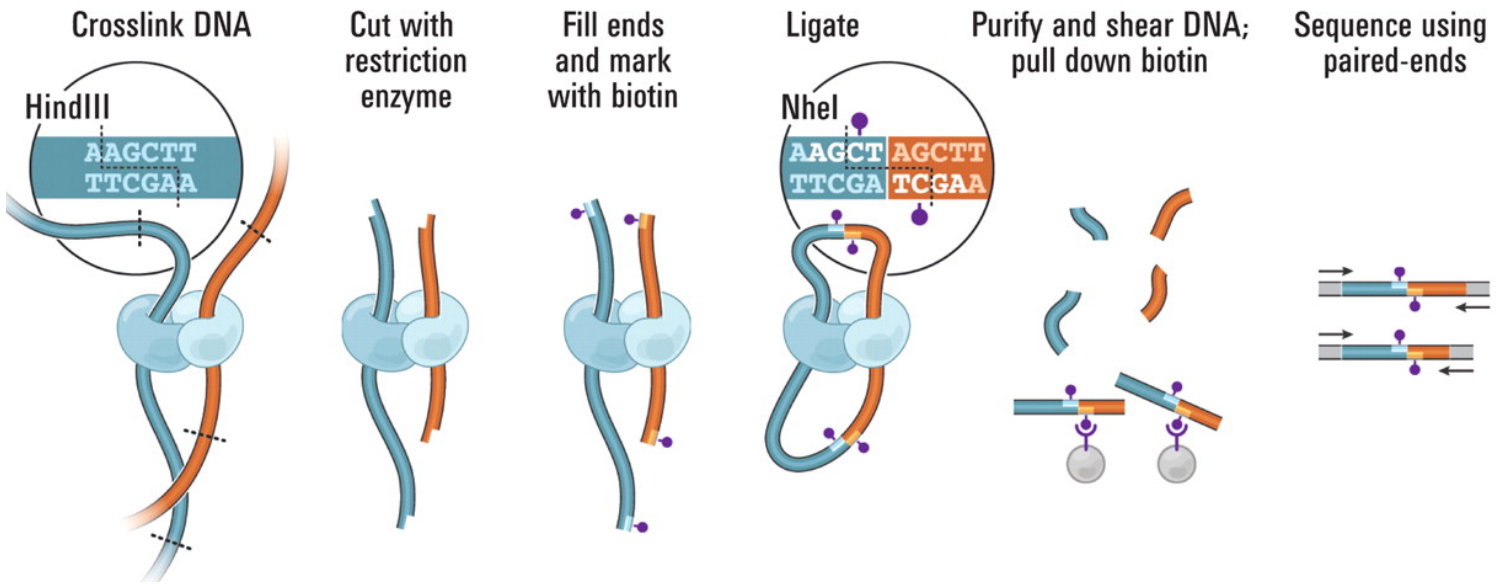
- RNA-Seq
- Chromosome conformation capture
- Metagenomics
- Many many others…

RNA-Seq: reverse transcribe RNA -> cDNA, sequence and count



- Computational/statistical tasks?
  - align and count reads
  - ID splice sites, splice variants
  - get normalized gene or transcript abundances
  - test for differential expression
  - modeling of gene expression

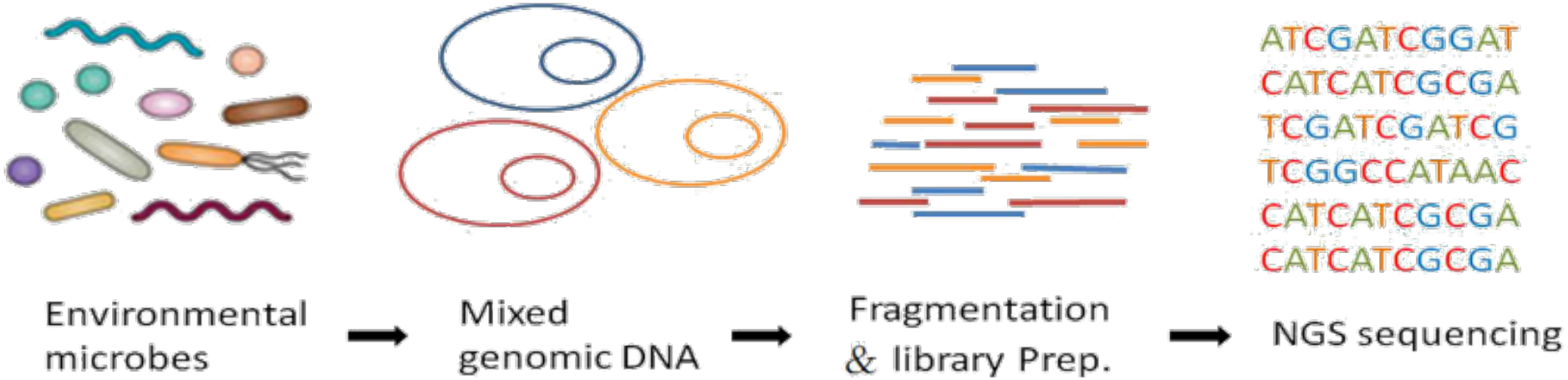
Chromosome conformation capture (3C, 5C, Hi-C)



- Computational/ statistical tasks?

- Identify ligation sites and count interactions
- Model physical structure based on contact map frequencies
- Test statistically for changes in conformation
- Relate to other data types

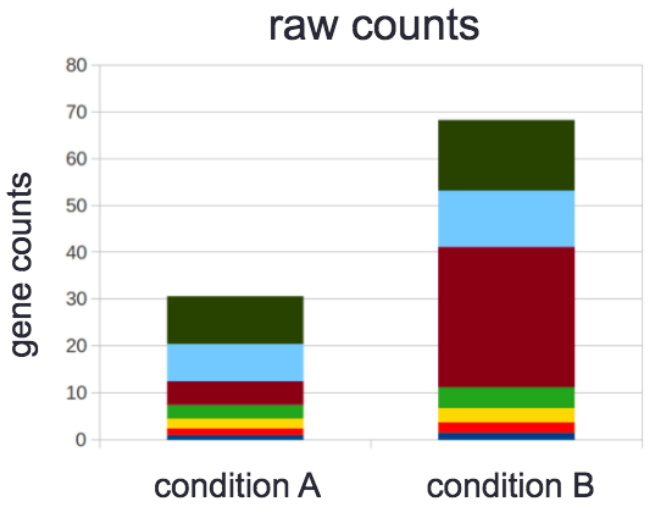
Whole metagenome shotgun sequencing



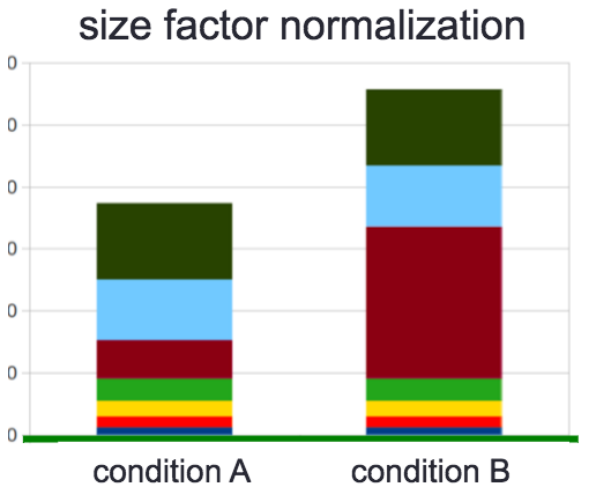
Computational/statistical tasks?

- Align reads to known genes and species
- Assemble genomes
- Quantify normalized abundances of species
- Look for genomic strain variation within a species (at the nucleotide and gene levels)
- Look for evidence of horizontal gene transfer events
- Quantify growth rate…?
- …

Normalizing data generated by sequencing assays is a surprisingly hard problem



Only red gene is truly differentially expressed

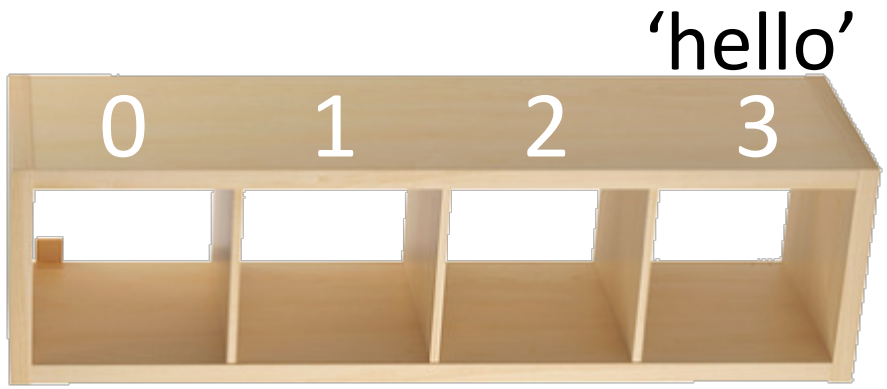


What is hashing?

- 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 – fast!!

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

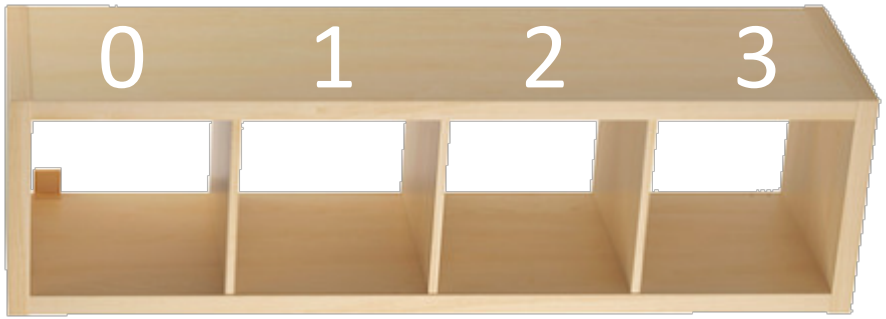
hash(‘goodbye’) → 3

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)

```
hash("ATGCTG") = key1
hash("TTTCTG") = key2
…
```

- **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 loction

Key	Hashed index	Genomic location
“GCTAGC”	Key1	Chr1 123412
…	…	…
“TTTAGC”	KeyN	Chr6 988472

Create a hash table that maps all observed 4-mers to its position(s) in the reference genome ‘s’

```
reference =
'ACAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCT'
k = 4 # size
h = {} # 'ACAA':[0], 'CCCC':[15,16]
```

Create a hash table that maps all observed 4-mers to its position(s) in the reference genome ‘s’

```
reference =
'ACAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCT'
k = 4 # size
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
```

```
print h

{'CGGC': [19], 'ACAA': [0], 'GTCC': [13], 'GGCC':
[20], 'AAGA': [2], 'TTGT': [11], 'ATTG': [10], 'CCGG':
[18], 'AGAT': [3], 'GATG': [4], 'ATGC': [5], 'GCTC':
[37], 'GCCA': [7], 'CAAG': [1], 'CCAT': [8], 'CCCC':
[15, 16], 'TGCC': [6], 'GCCT': [21], 'CCCG': [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()?

```
# 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?

```
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: 0.044001487732
```

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