Exercises

1. What is the time and memory complexity of the following algorithms?

Time: O(n)
Mem: O(1)

```
FunctionFactors(n):
    For i = 1 to n
        x = n*n
        print ‘x’
```

Time: O(n^2)
Mem: O(1)

```
FunctionCombination2(n):
    For i = 1 to n-1
        For j = i to n
            print ‘i, j’
```

Time: O(nm)
Mem: O(1)

```
FunctionCombination3(n, m):
    For i = 1 to n
        For j = 1 to m
            print ‘i, j’
        For k = 1 to m
            print ‘i, k’
```

Time: O(mn)
Mem: O(n+m)

```
StringSearch(n, m):
    For i = 1 to length(m)- length(n)
        match = 0
        For j = 1 to length(n)
            if (n[j] = m[i+j-1])
                match = match + 1
        If match > length(n)* 0.95
            print 'i'
```

Time: O(nm)
Mem: O(nm)

```
MatchMatrix(n, m):
    For i = 1 to length(n)
        For j = 1 to length(m)
            if (n[i] = m[j])
                matrix[i,j] = 1
            else
                matrix[i,j] = 0
```

2. Which of the following types of algorithms are inherently approximate? Which are stochastic?

approximate: greedy, randomization is often, but not always (quicksort)
stochastic: randomization, others can be stochastic but not necessarily

3. Why would you use branch and bound over exhaustive? faster or less memory

Today's objectives

- Evaluating strategies for short read alignment

String search

- Linear
- Hash
- Suffix tree
- Suffix array

- How does short read alignment work
- Burrows-Wheeler transform

Alignments covered & history

Classes

Pairwise vs multiple alignment
Local vs global alignment
Exhaustive vs approximate

Global, exhaustive, pairwise alignment
– Needleman S.B. and Wunsch C.D. (1970) J. Mol. Biol. 48, 443-453

Local, exhaustive, pairwise alignment
– Smith T.F. and Waterman M.S. (1981) J. Mol. Biol.147, 195-197

Local, approximate, pairwise alignment
– BLAST: Basic Local Alignment Search Tool (Altschul et al. 1990).

Global, approximate, multiple alignment
– ClustalW: Thompson,J.D. et al. (1994) Nucleic Acids Res., 22, 4673–4680.

Local, approximate, pairwise alignment
– Bowtie/BWA: time/memory solution to short read mapping (2009/2009)

Alignments covered & history

Classes

Pairwise vs multiple alignment
Local vs global alignment
Exhaustive vs approximate

Global, exhaustive, pairwise alignment
– Needleman S.B. and Wunsch C.D. (1970) J. Mol. Biol. 48, 443-453

Local, exhaustive, pairwise alignment
– Smith T.F. and Waterman M.S. (1981) J. Mol. Biol.147, 195-197

Local, approximate, pairwise alignment
BLAST: Basic Local Alignment Search Tool (Altschul et al. 1990).

Global, approximate, multiple alignment
ClustalW: Thompson,J.D. et al. (1994) Nucleic Acids Res., 22, 4673–4680.

Local, approximate, pairwise alignment
Bowtie/BWA: time/memory solution to short read mapping (2009/2009)

Start with string search
-simplest
-require seeds
-find substring
-fast/memory

A simple string search

Problem: Find all substrings (x) in a string (s)

Solution: Check each position for a match

```
S = ACTGACTGTA
x = CTG

P1  ACT
P2   CTG
P3     GAC
P4      ACT
P5       CTG
P6        TGT
P7         GTA
```

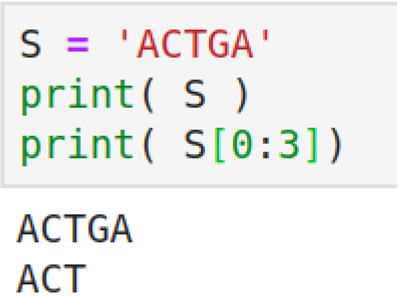
A simple string search

Problem: Find all substrings (x) in a string (s)

Solution: Check each position for a match

```
S = ACTGACTGTA
x = CTG

P1  ACT
P2   CTG
P3     GAC
P4      ACT
P5       CTG
P6        TGT
P7         GTA
```

```
stringsearch( x, S)
    lx = length( x )
    lS = length( S )
    for i in 1 to lS - lx:
        if x = S[i:i+lx]
            print( i )
```

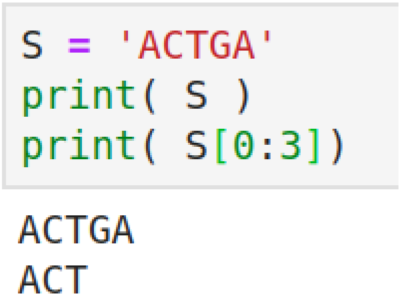


```
S = 'ACTGA'
print( S )
print( S[0:3])
```

```
ACTGA
ACT
```

S is a string; we can retrieve slices or indexed positions

String vs List

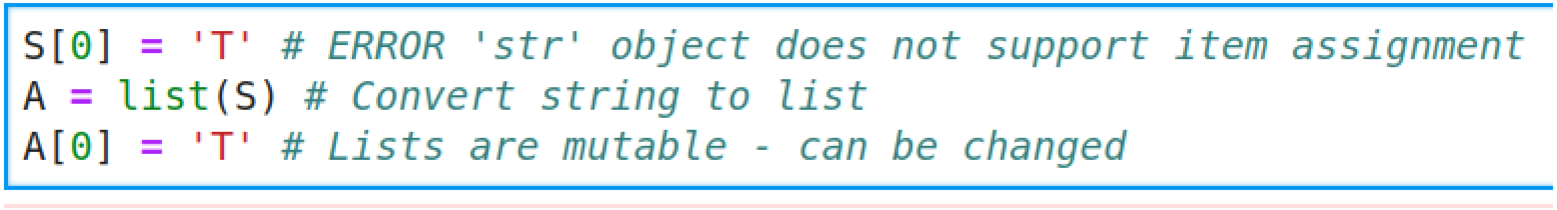


```
S = 'ACTGA'
print( S )
print( S[0:3])
```

```
ACTGA
ACT
```

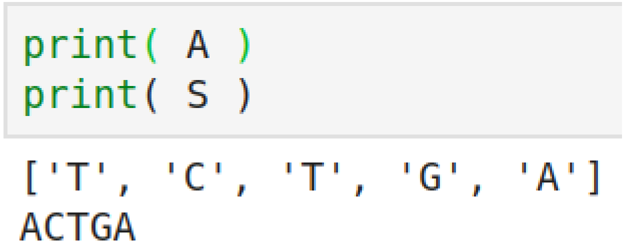
S is a string; we can retrieve slices or indexed positions

A is a list; we can retrieve indexed positions AND change the items



```
S[0] = 'T' # ERROR 'str' object does not support item assignment
A = list(S) # Convert string to list
A[0] = 'T' # Lists are mutable - can be changed
```

Linear search can be done with a string or list



```
print( A )
print( S )
```

```
['T', 'C', 'T', 'G', 'A']
ACTGA
```

A simple string search

Problem: Find all substrings (x) in a string (s)

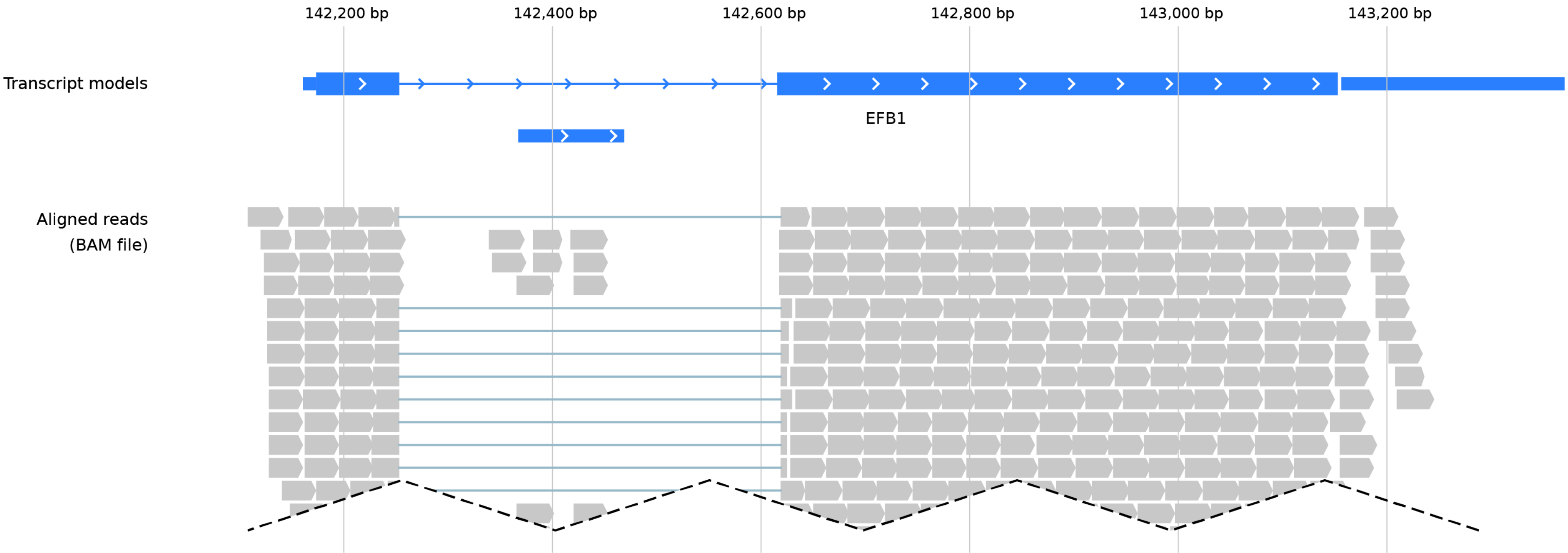
Solution: Check each position for a match

```
S = ACTGACTGTA
x = CTG

P1  ACT
P2   CTG
P3     GAC
P4      ACT
P5       CTG
P6        TGT
P7         GTA
```

```
stringsearch( x, S)
    lx = length( x )
    lS = length( S )
    for i in 1 to lS - lx:
        if x = S[i:i+lx]
            print( i )
```

Problem --- Complexity
Time: O(length(S))
Memory: O(length(S))

Short read alignment problem



Reads

ATCGATC
TCGATCG
CAGTTGC
CGGCTCG
AATATCT

Genome

GCGCTATCGGCTCGGCTCGGATCGGCAGTTGCGTA

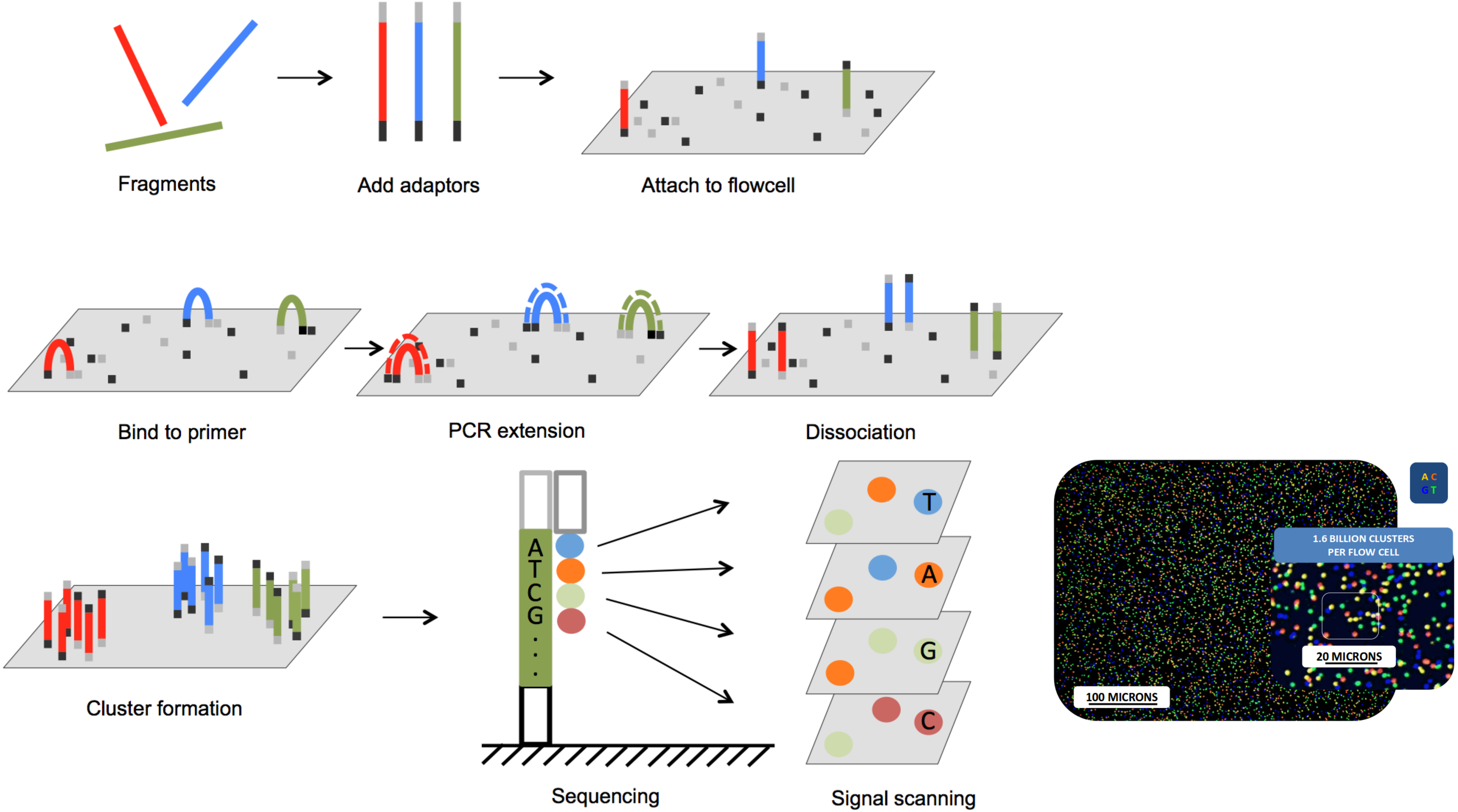
CGGCTCG
CGGCTCG
CAGTTGC

Goal: Find all exact matches of reads to genome
The problem: 10^6 reads and 10^6 to 10^9 genome size
(Reads are now 100 bp, used to be 30)
Time: O(genome size * reads) = 10^12
1e6/s = 10^6 seconds or 277 hours / million reads

Reads/genome
substring/string
words/book

Illumina Sequencing

Machine	reads/run
Illumina NovaSeq	10 billion
Illumina HiSeq X	6 billion



Fragments

Add adaptors

Attach to flowcell

Bind to primer

PCR extension

Dissociation

Cluster formation

Sequencing

Signal scanning

Today's objectives

- Evaluating strategies for short read alignment

String search

- Linear
- Hash
- Suffix tree
- Suffix array

- How does short read alignment work
- Burrows-Wheeler transform

Algorithm ~ Data structure

- Linear search (array ie Python list)
- Hash (ie Python dictionary)

Algorithm History

Algorithm ~ Data structure

- Linear search (array ie Python list)
- Hash (ie Python dictionary)

Suffix Tree

- Weiner (1973)
- Knuth (father of algorithm analysis) declared it algorithm of the year

Suffix Array

- Manber and Myers (1990)
- space efficient alternative to suffix tree.

Burrows-Wheeler Transform

- Discovered by David Wheeler in 1983, published by Michael Burrows and David Wheeler in 1994
- reversible
- compression

FM-index

- Paolo Ferragina and Giovanni Manzini (2000)
- compressed full-text substring index based on the Burrows-Wheeler transform. It is used to efficiently find the number of occurrences of a pattern within the compressed text.

Short Read Alignment

Problem: Where in the genome did this short (30-100 bp) sequence come from?

- Tolerate mismatches (errors)
- Map millions of reads per hour
- Memory 2.7 GB human genome (770 Mb)
  - 1 bp = 2 bits (0/1), 1 byte = 8 bits

Solution: Burrows-Wheeler transform with FM-index

- Bowtie, BWA, SOAP2
- 25 million 35-bp reads per hour
- 1.3 GB memory

Linear search: naive, brute force

```
CAGATCTGCATGCATCGTAGCTAGCTACGATCGTCG
ATGC
 ATGC
  ATGC
   ATGC
    ATGC
     ATGC
      ATGC
       ATGC
        ATGC
         ATGC
```

What's a faster way to do this?

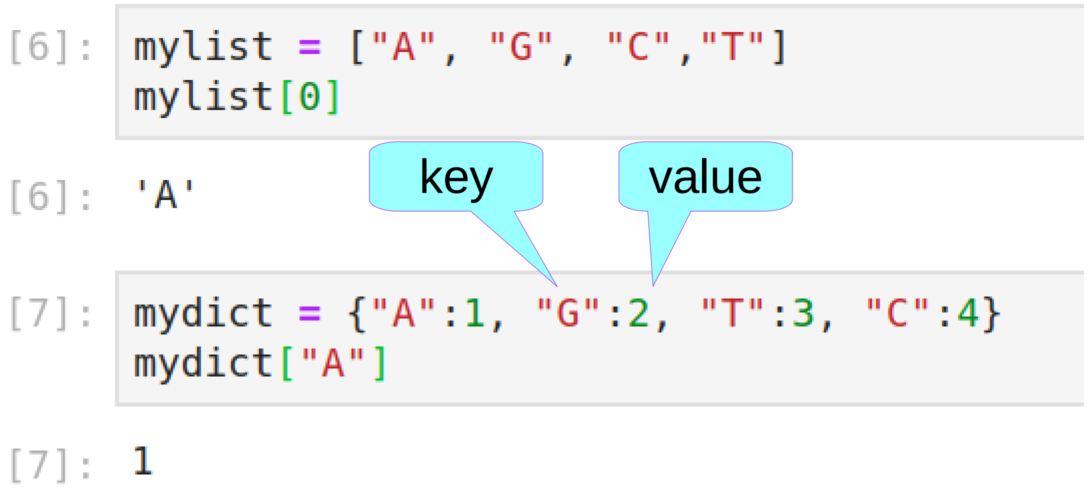
What about a hash table?

Can we look for inexact strings? YES

Hash Table

A hash table is a data structure which implements an associative array abstract data type, a structure that can map keys to values.

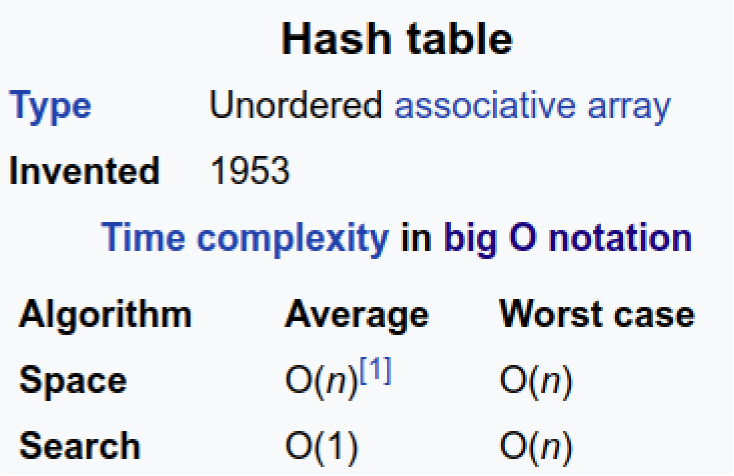
Python's Dictionary is implemented using a hash table, indexed by keys rather than numbers (List)



```
[6]: mylist = ["A", "G", "C","T"]
     mylist[0]

[6]: 'A'

[7]: mydict = {"A":1, "G":2, "T":3, "C":4}
     mydict["A"]

[7]: 1
```

Hash Table

A hash table is a data structure which implements an associative array abstract data type, a structure that can map keys to values.

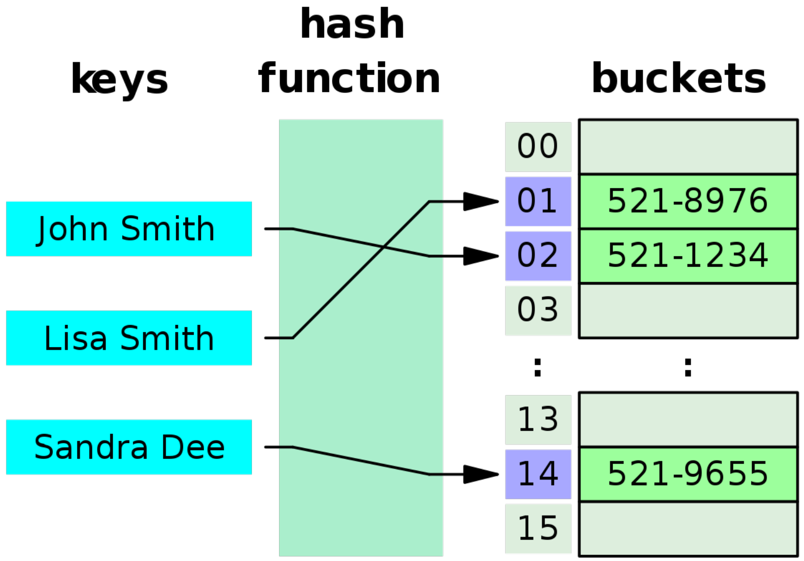


Hash table

Type	Unordered associative array
Invented	1953

Time complexity in big O notation

Algorithm	Average	Worst case
Space	$O(n)$[1]	$O(n)$
Search	$O(1)$	$O(n)$



Hash collisions occur when the hash function generates the same index for more than one key. Collisions are unavoidable when hashing a large set of keys. If 2,450 keys are hashed into a million buckets, there is approximately a 95% chance of at least two of the keys being hashed to the same slot (birthday problem).

Don't worry about collisions: Python etc take care of this for you.

Hash table

S = 'AGCCCGG'

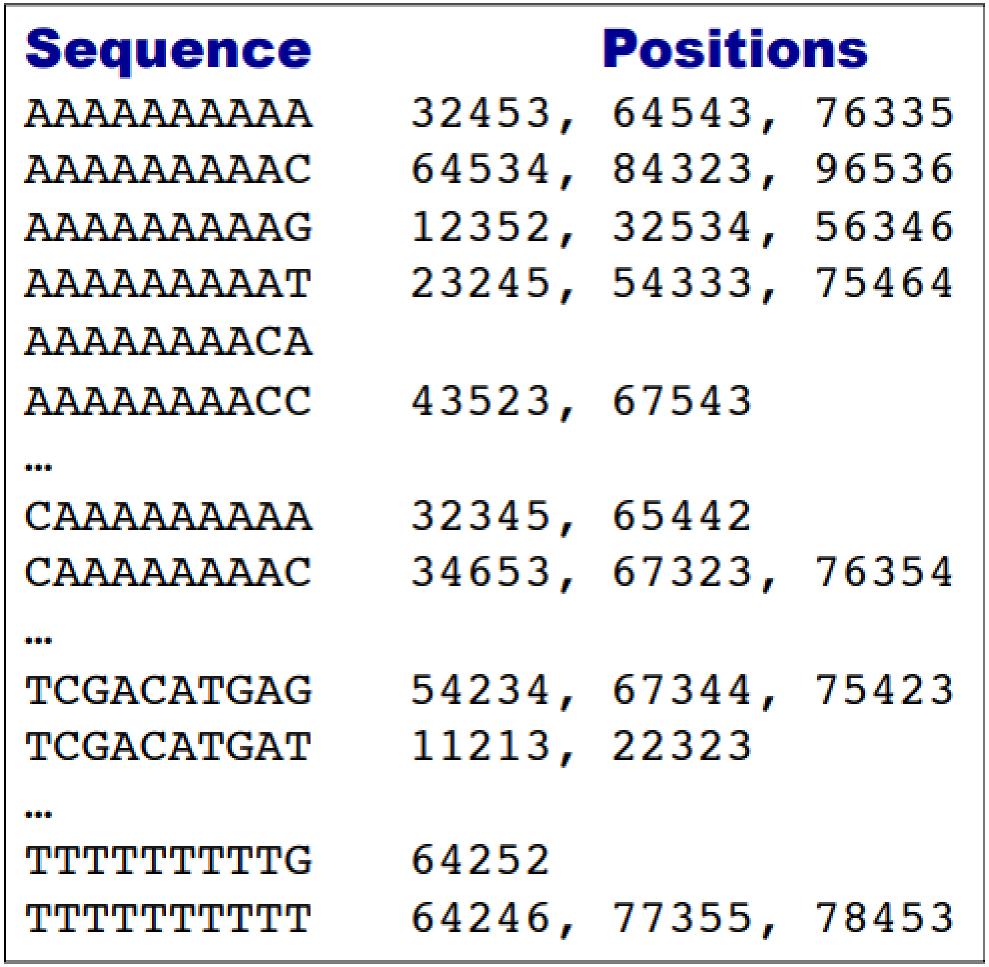
Key	Value
AG	0
GC	1
CC	2,3
CG	4
GG	5

Creating hash table
Time O(n)
Memory O(n)

Searching hash table
Time O(1)
Memory O(n)

How does complexity increase with # searches (m)?
Creating: O(1)
Searching: O(m)

Problem #1: multiple positions

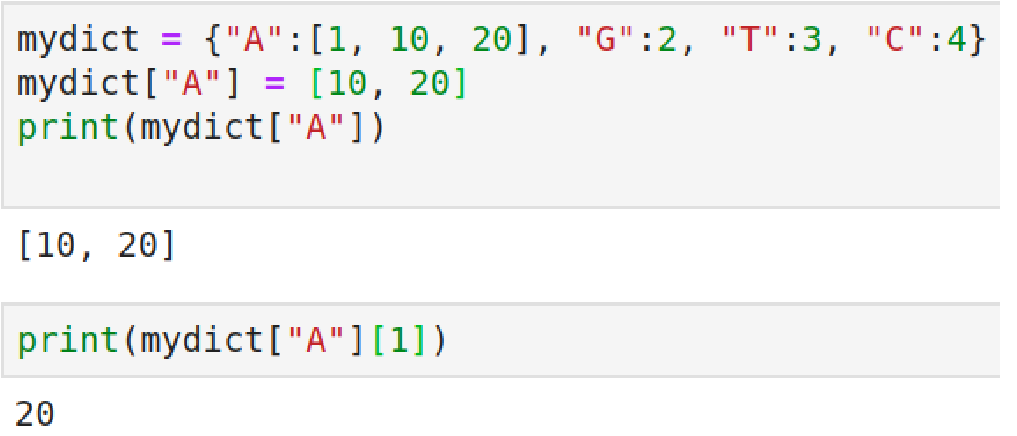


Sequence	Positions
AAAAAAAAAA	32453, 64543, 76335
AAAAAAAAAC	64534, 84323, 96536
AAAAAAAAAG	12352, 32534, 56346
AAAAAAAAAT	23245, 54333, 75464
AAAAAAAACA	
AAAAAAAACC	43523, 67543
…	
CAAAAAAAAA	32345, 65442
CAAAAAAAAC	34653, 67323, 76354
…	
TCGACATGAG	54234, 67344, 75423
TCGACATGAT	11213, 22323
…	
TTTTTTTTTG	64252
TTTTTTTTTT	64246, 77355, 78453

N = length of genome
L = string length
4^L = possible strings
$N/4^L$ positions/string

If L = 10, N=3e9 or 3×10^9:
positions/string ~ 3,000

Solution: Dictionary of lists



```
mydict = {"A":[1, 10, 20], "G":2, "T":3, "C":4}
mydict["A"] = [10, 20]
print(mydict["A"])
```

```
[10, 20]
```

```
print(mydict["A"][1])
```

```
20
```

Problem #2: memory

N = length of genome
L = string length
4^L = possible strings
$N/4^L$ positions/string

Humans
3.2 Gbp genome = 3e9

If L = 10, N=3e9, ~3000 positions, slower seeding
If L = 15, there will be ~ 3 positions, faster seeding
But... requires 10^9 (4^{15}) hash size

1 bit = 0/1
1 byte = 8 bits
1 character = 8 bits (1 byte)
1 integer = 32 bits (4 bytes)
1 double (float) = 64 bits (8 bytes)

hash keys (L=15):
10^9 * 15 bytes = 15 Gb
hash values:
10^9 * 4*3 bytes = 12 Gb
Trade: Memory ~ Speed

1 integer * 3 positions

Problem #3: fixed string length

Suppose sequences aren't random
L = 15 but reads are typically 100 bp
Expected # positions = 3
But observed # positions = between 0 and 1000

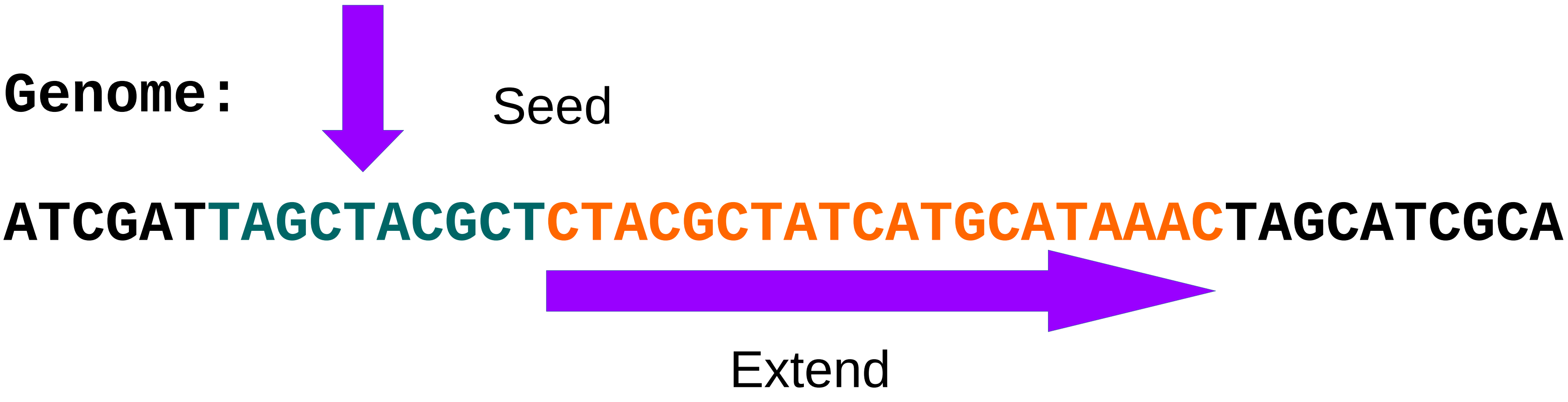
Solution #1
seed and extend, but slow if 1000 positions to check

Problem #3: fixed string length

Suppose sequences aren't random
L = 15 but reads are typically 100 bp
Expected # positions = 3
But observed # positions = between 0 and 1000

Solution #1
seed and extend, but slow if 1000 positions to check

```
Read:  TAGCTACGCTCTACGCTATCATGCATAAAC
Seed:  TAGCTACGCT
```



Genome:

Seed

```
ATCGATTAGCTACGCTCTACGCTATCATGCATAAACTAGCATCGCA
```

Extend

Problem #3: fixed string length

Suppose sequences aren't random
L = 15 but reads are typically 100 bp
Expected # positions = 3
But observed # positions = between 0 and 1000

Solution #1
seed and extend, but slow if 1000 positions to check

Solution #2
Longer hashes, but requires more memory and can't handle errors (string mismatches)

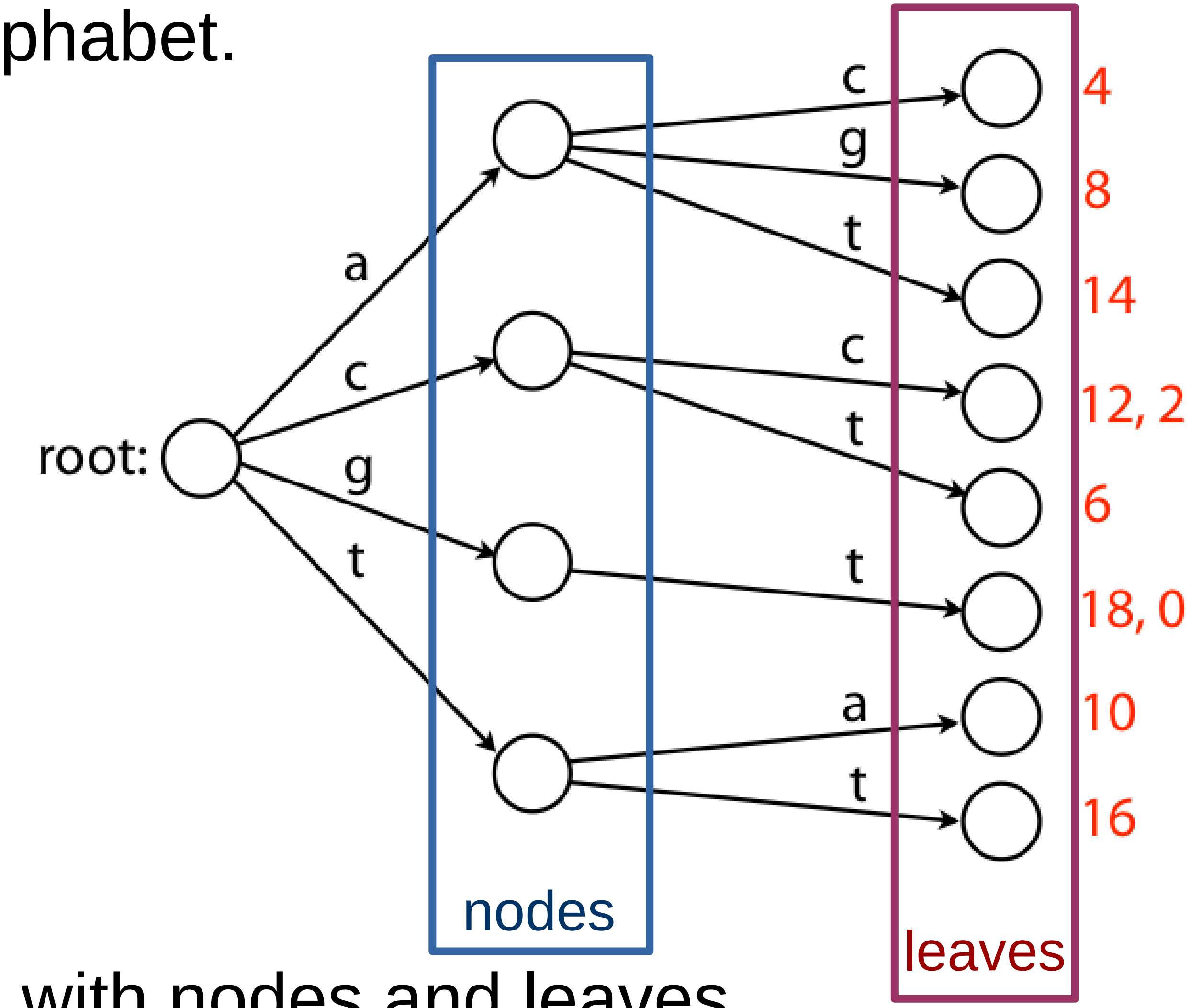
Data Structures

Array (Python list) linear search, slow, errors	Time O(n), Memory O(n)
Hash (Python dictionary) lookup table, memory, fixed length, fast	Time O(1) Memory O(n)
Suffix tree (Mummer)	
Suffix array	
Burrows-Wheeler transform and FM index (BWA and Bowtie)	

Suffix Tree and Trie

Trie (radix tree or prefix tree) is a kind of search tree, an ordered tree data structure that is used to store strings over an alphabet.

ac	4
ag	8
at	14
cc	12
cc	2
ct	6
gt	18
gt	0
ta	10
tt	16



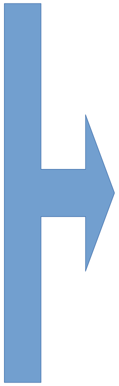
- root is empty
- keys are associated with nodes and leaves
- values are associated with leaves

Suffix trie

Suffix trie helpful to understand suffix tree

Suffix trie: build a trie containing all suffixes of text T

```
T: abaaba
   abaaba
    baaba
     aaba
      aba
       ba
        a
```



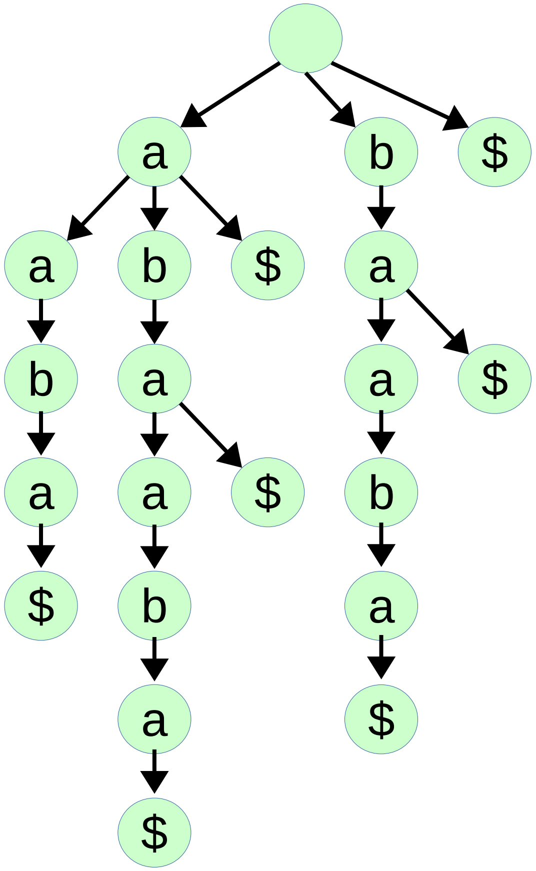
Suffixes
n(n+1)/2 characters

Suffix trie

Suffix trie: build a trie containing all suffixes of text T

- Add terminal character $ to the end of T
  - $ < A < C < G < T
  - ab comes before aba since ab$ < aba$
  - no suffix is a prefix for any other suffix

```
T:  abaaba$
    abaaba$
     baaba$
      aaba$
       aba$
        ba$
         a$
          $
```



Suffix trie

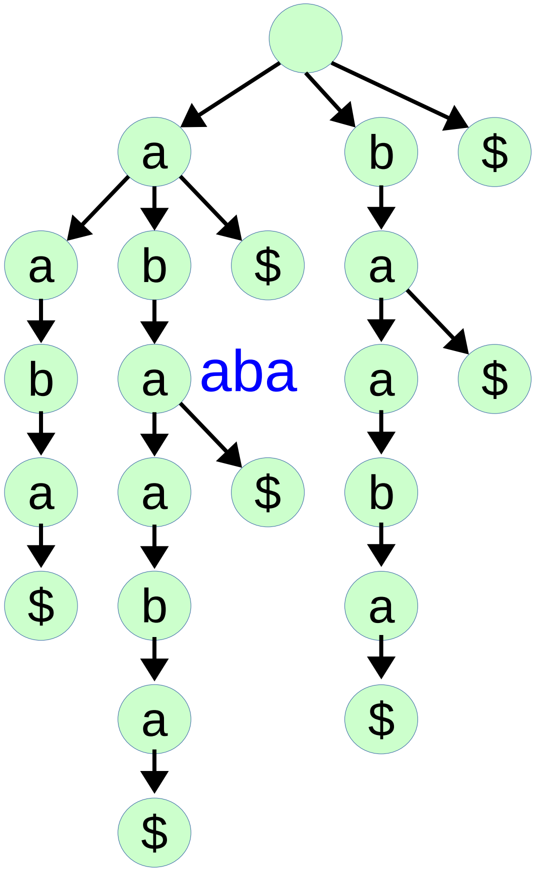
Suffix trie: build a trie containing all suffixes of text T

- Add terminal character $ to the end of T
- $ < A < C < G < T
- ab comes before aba since ab$ < aba$
- no suffix is a prefix for any other suffix

```
T: abaaba$
   abaaba$
    baaba$
     aaba$
      aba$
       ba$
        a$
         $
```

Does string 'aba' exist?
What about 'abba'?
How many times does 'aba' occur?
Where is 'aba'

n < nodes < n^2



Suffix Tree

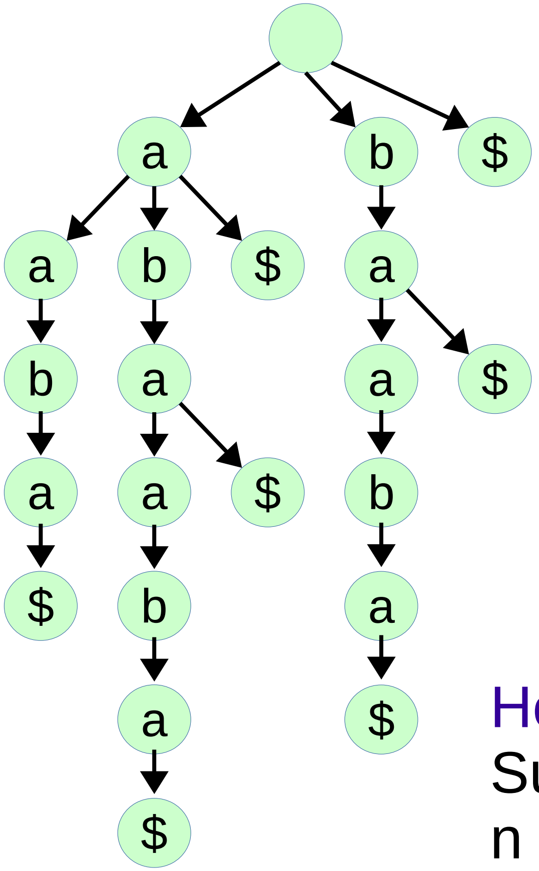
- Suffix Tree: a compressed trie containing all the suffixes of the given text as their keys and positions in the text as their values

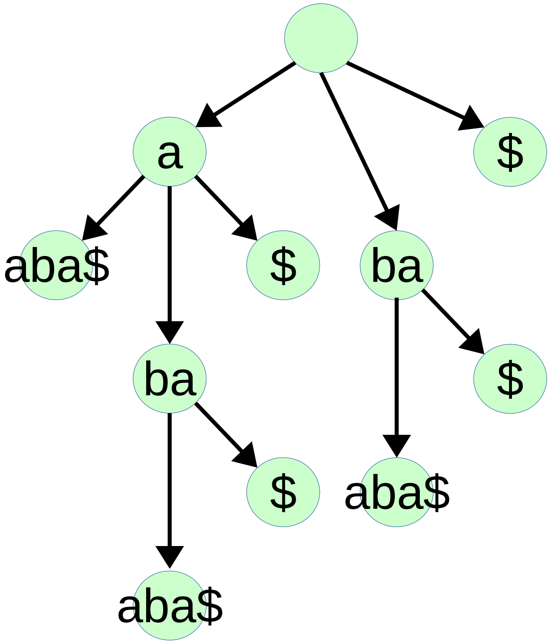
- Suffix trees provide linear-time solutions for the longest common substring problem.

- Cost: storing a string's suffix tree typically requires significantly more space than storing the string itself.

- Mummer: Delcher et al. (1999), Nucleic Acids Research, 27:2369-2376.

Suffix tree: compressed trie





How many nodes?

Suffix trie: n(n+1)/2

n leaf nodes

n-1 nodes (non-leaf)

2n-1 = O(n) (Suffix tree)

But suffix tree still needs n^2 characters

keys can be stored using T and (offset, length)

Suffix tree construction

O(n) time (Ukkonen)

O(n) memory

Search

O(m) m= # substrings

Suffix Array

Suffix array is an array of integers specifying the lexicographic ordering of the suffixes of a string.

As only positions are stored, it is a simple, space efficient alternative to suffix trees.

Suffix	i
abaaba$	1
baaba$	2
aaba$	3
aba$	4
ba$	5
a$	6
$	7

sorted	i
$	7
a$	6
aaba$	3
aba$	4
abaaba$	1
ba$	5
baaba$	2

SA = [7,6,3,4,1,5,2]

Where is 'ba'?
Use binary search.

Binary Search

Binary search: finds the position of a target value within a sorted array.

Time: average O(log n)

'ba' search

sorted	i
$	7
a$	6
aaba$	3
aba$	4
abaaba$	1
ba$	5
baaba$	2

L

←

R

- Binary search compares the target value to the middle element of the array.
- If the target is greater than the midpoint the target cannot be to the left and so the search continues on the right
- The midpoint of the right side is found, queried against the target and this is repeated until the target is found

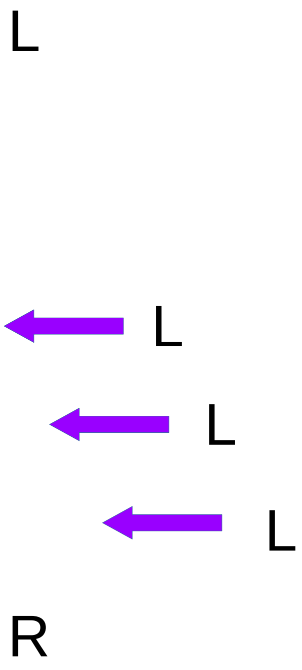
Binary Search

Binary search: finds the position of a target value within a sorted array.

Time: average O(log n)

'ba' search

sorted	i
$	7
a$	6
aaba$	3
aba$	4
abaaba$	1
ba$	5
baaba$	2



L

L

L

L

R

Variations (bisect left or right)

- round the midpoint to the lower (floor) or upper (ceil) integer
- is 'ba' absent
- is 'ba' present multiple times

Data Structures

Array (Python list) linear search, slow, errors	Time O(n), Memory O(n)
Hash (Python dictionary) lookup table, memory, fixed length, fast	Time O(1) Memory O(n)
Suffix tree (Mummer), memory, variable length, fast	Time O(m) Memory O(n) Construction O(n)
Suffix array, memory, variable length, fast	Time O(m log(n)) Mem O(n) non-Naive O(m)
Burrows-Wheeler transform and FM index (BWA and Bowtie)	

Speed vs Memory tradeoff
Suffix tree: a bit more memory ~ 20n, but faster than Suffix array

Time vs Memory

Brute Force
(3 GB)

```
BANANA
BAN
 ANA
  NAN
   ANA
```

Naive

Slow & Easy

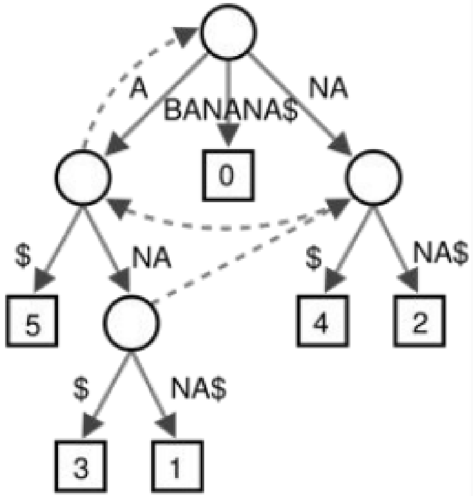
Suffix Array
(>15 GB)

6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Vmatch, PacBio Aligner

Binary Search

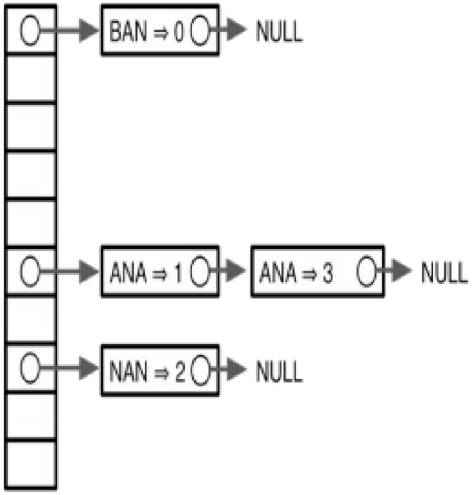
Suffix Tree
(>51 GB)



MUMmer, MUMmerGPU

Tree Searching

Hash Table
(>15 GB)



BLAST, MAQ, ZOOM, RMAP, CloudBurst

Seed-and-extend

Compressed Suffix array
- Burrows-Wheeler Transform
- FM-index

Exercises

1) For each mapping approach:

   i) Why not use the naive, brute force sliding window approach?

   ii) Why not use hashes?

   iii) Why not use suffix trees?

   iv) Why not use suffix arrays?

2) Why are data structures important for the string search problem?

3) Which data structure would you use to search for substrings of varying lengths? (hash, suffix trees, suffix arrays)