

# COMBINATORIAL PATTERN MATCHING

# EXACT MATCHING

- Tabulating patterns in long texts
  - Short patterns (direct indexing)
  - Longer patterns (hash tables)
- Finding exact patterns in a text
  - Brute force (run time)
  - Pattern preprocessing (prefix trees)
  - Text preprocessing (suffix trees)

# APPROXIMATE MATCHING

- Algorithms for approximate pattern matching
- Heuristics behind BLAST
- Statistics behind BLAST

# STRING ENCODING

- It is often necessary to index strings; a convenient way to do this is first to convert strings to integers.
- Given a string  $s$  of length  $n$  on alphabet  $\mathbf{A}$  ( $0..c-1$ ), with  $c=|\mathbf{A}|$  characters, we can define a map  $\text{code}(s) \rightarrow [0, \infty)$ , as
$$\text{code}(s) \rightarrow s[1]c^{n-1} + s[2]c^{n-2} + \dots + s[n-1]c + s[n]$$
- There are  $c^L$  different  $L$ -mers, but at most  $n-L+1$  different  $L$ -mers in a text of length  $n$

A	0
C	1
G	2
T	3

AGT	A=0*16	G=2*4	T=3	11
ATA	A=0*16	T=3*4	A=0	12
TGG	T=3*16	G=2*4	G=2	58

## TABULATING SHORT PATTERNS

- If the L is small (e.g. 3 or 4) use direct indexing to tabulate strings efficiently

AAA = 0, AAC = 1, AAG = 2... TTT = 64

direct indexing of the  
list **a** = (2,2,5,6,8,9,11)

**b** = (0,0,1,0,0,1,1,0,1,1,0,1)

what about (2,55,10<sup>34</sup>)?

```
INDEXLIST(a,n)
M = max (a)
for i = 0 to M
  bi = 0
for i = 0 to n
  bai = 1
return b
```

## TABULATING SHORT PATTERNS

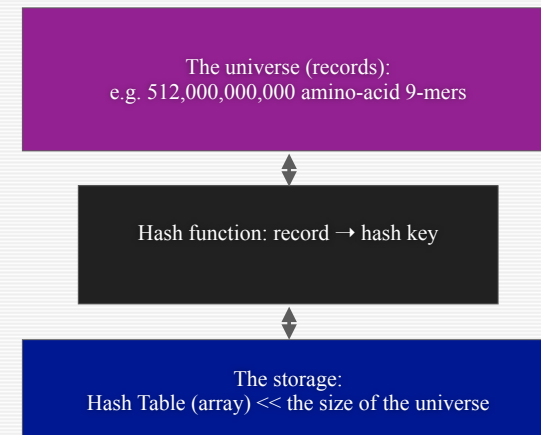
- The distribution of short strings in genetic sequences is biologically informative, e.g.
- Synonymous codons (triplets of nucleotides, 64 patterns) are often used preferentially in organisms (transcriptional selection, secondary structure, etc)
- The location of short amino-acid strings (e.g. L=3, 8000 patterns) is useful for finding seeds for BLAST

## TABULATING/LOCATING LONGER PATTERNS

- Finding motifs
  - Motifs: promoter regions, functional sites, immune targets
  - Cellular immunity targets in pathogens (e.g. protein 9 mers)
- There are too many patterns to store in an array, and even if we could, then the array would be very sparse
  - E.g. ~512 billion (20<sup>9</sup>) amino-acid 9-mers, but in an average HIV-1 sequence (~3000 amino acids long) there are at most ~3000 unique 9-mers

## HASH TABLES

- Allow to efficiently store and retrieve a small subset of a large universe of records. Hash tables implement associative arrays in a variety of languages (Python, Perl etc)



# A SIMPLE HASH FUNCTION

- A reasonable hash function (on integer records  $i$ ) is:  $i \rightarrow i \bmod P$
- $P$  is a prime number and also the natural size of the hash table (minimizes the number of 4-mers mapping to 0)
- Hash keys range from 0 to  $P-1$
- If the records are uniformly distributed, so will be their hash keys

$P=101$

4-mer (256 possible)	Integer code	Hash Key
ACGT	27	27
CCCA	148	47
TGCC	229	27

COLLISION

# COLLISIONS

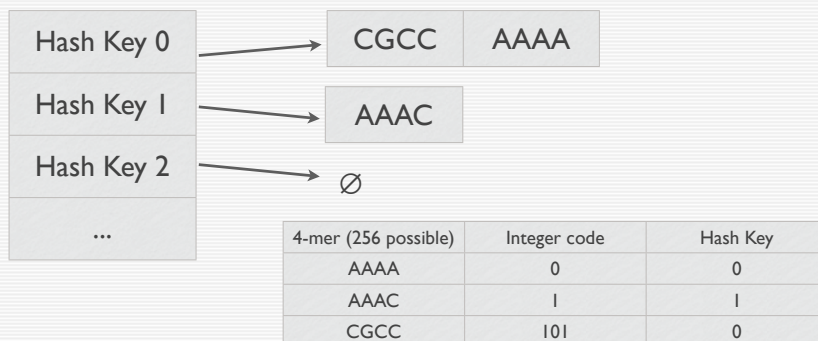
- Collisions are frequent even for sparsely populated lightly loaded hash tables
  - load level  $\alpha = (\text{number of entries in hash table})/(\text{table size})$
- The birthday paradox: what is the probability that two people out of a random group of  $n$  ( $<365$ ) people share a birthday (in hash table terms, what is the probability of a collision if people=records and hash keys=birthdays)?

$$P(n) = 1 - \left(1 - \frac{1}{365}\right)\left(1 - \frac{2}{365}\right) \dots \left(1 - \frac{n-1}{365}\right)$$

n	$\alpha$	P(n)
10	0.027	0.117
23	0.063	0.507
50	0.137	0.97

# DEALING WITH COLLISIONS

- use chaining
  - Each hash key is associated with a linked list of all records sharing the hash key



# EXACT PATTERN MATCHING

- Motivation: Searching a database for a known pattern
- Goal: Find all occurrences of a pattern in a text
- Input: Pattern  $P = p[1] \dots p[n]$  and text  $T = t[1] \dots t[m]$  ( $n \leq m$ )
- Output: All positions  $1 < i < (m - n + 1)$  such that the  $n$ -letter substring of text  $T[i][i+n-1]$  starting at  $i$  matches the pattern  $P$
- Desired performance:  $O(n+m)$

# BRUTE FORCE PATTERN MATCHING

**Data** : Pattern P, Text T

**Result**: The list of positions in T where P occurs

```

1 n ← len(P);
2 m ← len(T);
3 for i:=1 to m-n+1 do
4   | if T[i:i+n-1] = P then
5   |   output i;
6   | end
7 end
    
```

Substring comparison can take from 1 to n (left-to-right) string comparisons

□ Text: **GGCATC**; Pattern: **GCAT**

i=1 (2 comparisons)

G	G	C	A	T	C
G	C	A	T		

i=2 (4 comparisons)

G	G	C	A	T	C
	G	C	A	T	

i=3 (1 comparison)

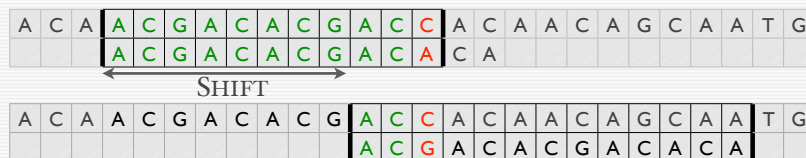
G	G	C	A	T	C
		G	C	A	T

# BRUTE FORCE RUN TIME

- Worst case:  $O(nm)$ . eg: searching for  $P=AA...C$  in text  $T=AA...A$ ,
- Expected on random text:  $O(m)$
- 1st pos match:  $\frac{1}{A}$  1st pos mis-match  $\frac{A-1}{A}$
- 1st match & 2nd pos mismatch:  $(\frac{1}{A})(\frac{A-1}{A}) = \frac{A-1}{A^2}$
- up to n-1th pos match and nth pos mismatch:  $\frac{A-1}{A^n}$
- Genetic texts are not random, so the performance may degrade.

# IMPROVING THE RUN TIME

- The search pattern can be preprocessed in  $O(n)$  time to eliminate backtracking in the text and hence guarantee  $O(n+m)$  run time
- A variety of procedures, starting with the Knuth-Morris-Pratt algorithm in 1977, take this approach. Makes use of the observation that if a string comparison fails at pattern position  $i$ , then we can shift the pattern  $i-b(i)$  positions, where  $b(i)$  depends on the pattern and continue comparing at position the same or the next position in the text, thus avoiding backtracking.
- These types of algorithms are popular in text editors/mutable texts, because they do not require the preprocessing of (large) text

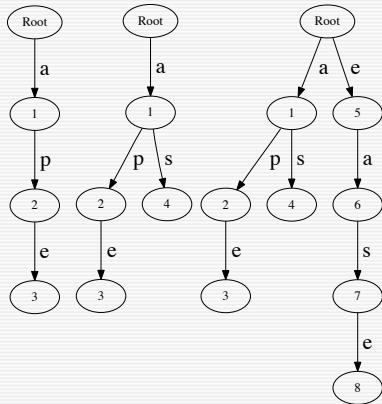


# EXACT MULTIPLE PATTERN MATCHING

- The problem: given a dictionary of  $D$  patterns  $P_1, P_2, \dots, P_D$  (total length  $n$ ) and text  $T$  report all occurrences of every pattern in the text.
- Arises, for instance when one is comparing multiple patterns against a database
- Assuming an efficient implementation of individual pattern comparison, this problem can be solved in  $O(Dm+n)$  time by scanning the text  $D$  times.
- Aho and Corasick (1975) showed how this can be done efficiently in  $O(m+n)$  time.
- Uses the idea of a trie (from the word retrieval), or prefix trie
- Intuitively, we can reduce the amount of work by exploiting repetitions in the patterns.

# PREFIX TRIE

- Patterns: 'ape', 'as', 'ease'. Constructed in  $O(n)$  time, one word at a time.

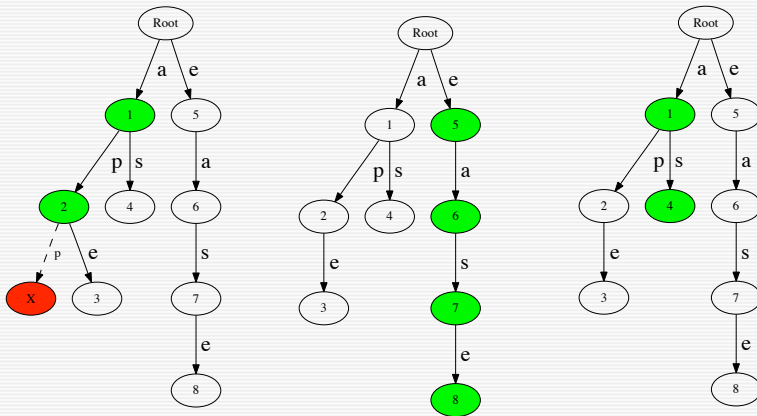


- Properties of a trie
  - Stores a set of words in a tree
  - Each **edge** is labeled with a letter
  - Each **node** labeled with a *state* (order of creation)
  - Any two edges sharing a parent node have distinct labels
  - Each word can be spelled by tracing a path from the root to a leaf

# SEARCHING TEXT FOR MULTIPLE PATTERNS USING A TRIE: THREADING

- Suppose we want to search the text **'appease'** for the occurrences of patterns 'ape', 'as' and 'ease', given their trie.
- The naive way to do it is to thread (i.e. spell the word using tree edges from the root) the text starting at position  $i$ , until either:
  - A leaf (or specially marked terminal node) is reached (a match has been found)
  - Spelling cannot be completed (no match)

start at:  $i=1$ : no match       $i=4$ : match       $i=5$ : Match  
 APPEASE                      APPEASE                      APPEASE



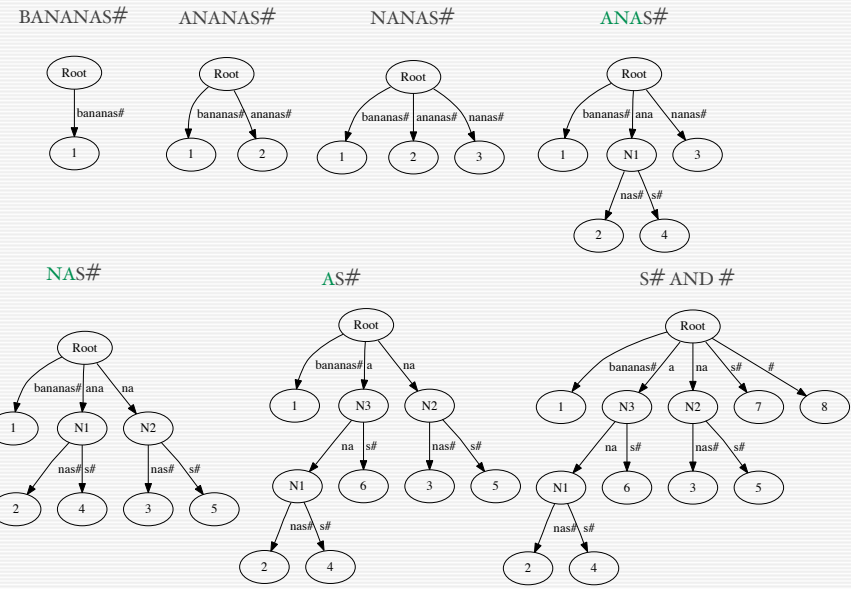
$O(m+n)$  time where  $m$  = length of text,  $n$  = total length of queries

# SUFFIX TREES

- A trie that is built on every suffix of a text  $T$  (length  $m$ ), and collapses all interior nodes that have a single child is called a **suffix tree**.
- A very powerful data structure, e.g. given a suffix tree and a pattern  $P$  (length  $n$ ), all  $k$  occurrences of  $P$  in  $T$  can be found in time  $O(n+k)$ , i.e. independently of the size of the text (but it figures into the construction cost of tree  $T$ )
- A suffix tree can be built in linear time  $O(m)$  i.e. length of text to search

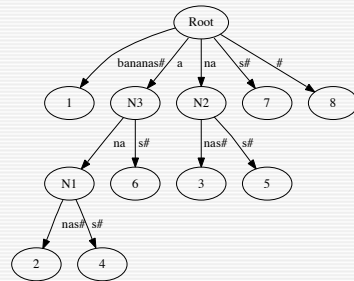
# BUILDING A SUFFIX TREE

- Example 'bananas#'. It is convenient to terminate the text with a special character, so that no suffix is a prefix of another suffix (e.g. as in *banana*). This guarantees that spelling any suffix from the root will end at a leaf.
- Construct the suffix tree in two phases from the longest to the shortest suffix:
  - Phase 1: Spell as much of the suffix from the root as possible
  - Phase 2: If stopped in the middle of an edge, break the edge and add a new branch spell the rest of the suffix along that branch. Label the leaf with the starting position of the suffix.



# SUFFIX TREE PROPERTIES

- Exactly  $m$  leaves for text of size  $m$  (counting the terminator)
- Each interior node has at least two children (except possibly the root); edges with the same parent spell substrings starting with different letters.
- The size of the tree is  $O(m)$
- Can be constructed in  $O(m)$  time
- This uses the observation that during construction, not every suffix has to be spelled all the way from the root (which would lead to quadratic time);
- Is also memory efficient (about  $\sim 5m \cdot \text{sizeof}$  (long) bytes for text without too much difficulty)

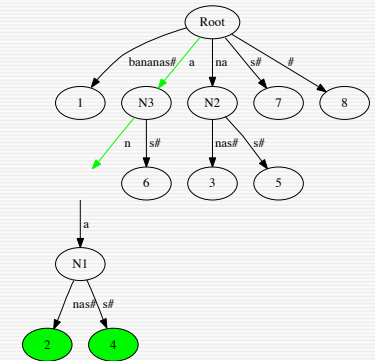


# MATCHING PATTERNS USING SUFFIX TREES

- Consider the problem of finding pattern 'an' in the text 'bananas#'

  - Two matches: positions 2 and 4

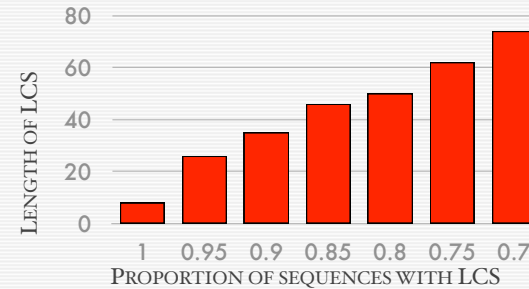
- Thread the pattern onto the tree
  - Completely spelled: report the index of every leaf below the point where spelling stopped. This is because the pattern is a prefix of every suffix spelled by traversing the rest of the subtree.
  - Incompletely spelled: no match
  - Runs in  $O(n+k)$  time, where  $n$  is the length of the pattern, and  $k$  is the number of matches.



# INEXACT PATTERN MATCHING

- Homologous biological sequences are unlikely to match exactly; evolution drives them apart with mutations for example.
- Pattern matching with errors is a fundamental problem in bioinformatics – finding homologs in a database.
- Well-performing heuristics are frequently used.

## EXAMPLE: LONGEST COMMON SUBSTRING (LCS) IN INFLUENZA A VIRUS (IAV) H5N1 HEMAGGLUTININ (N=957 FROM 2005+)



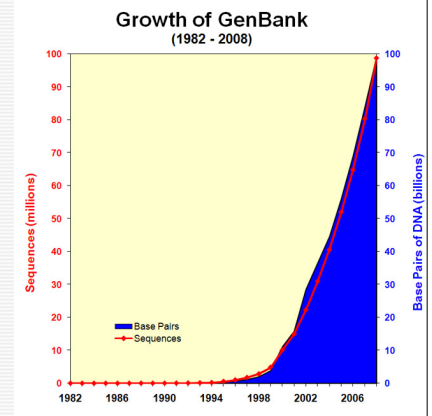
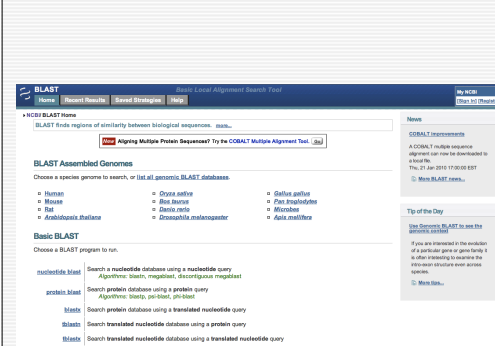
- Suffix trees can be adapted to efficiently find LCS from a proportion of a set of sequences as well.
- The longest fully conserved nucleotide substring in viruses sampled in 2005 or later is **merely 8 nucleotides long**
- This poses significant challenges for even straightforward tasks, such as diagnostic probe design

# K-DIFFERENCES MATCHING

- The k-mismatch problem: given a text **T** (length **m**), a pattern **P** (length **n**) and the maximum tolerable number of mismatches **k**, output all locations *i* in **T** where there are at most **k** differences between **P** and **T**[*i*:*i*+**n**-1]
- The k-differences problem: can also match characters to indels (cost 1) -- a generalization.
- Both can be easily solved in  $O(nm)$  time, by either brute force or dynamic programming (sequence alignment)

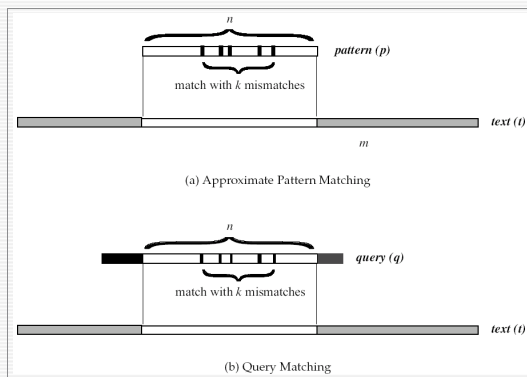
# GENBANK

[HTTP://WWW.NCBI.NLM.NIH.GOV/](http://www.ncbi.nlm.nih.gov/)



## QUERY MATCHING

- If the pattern is long (e.g. a new gene sequence), it may be beneficial to look for substrings of the pattern that approximately match the reference (e.g. all genes in GenBank).



## QUERY MATCHING

- Approximately matching strings share some perfectly matching substrings (**L**-mers).
- Instead of searching for approximately matching strings (**difficult, quadratic**) search for perfectly matching substrings (**easy, linear**).
- Extend obtained perfect matches to obtain longer approximate matches that are locally optimal.
- This is the idea behind probably the most important bioinformatics tool: **Basic Local Alignment Search Tool** (Altschul, S., Gish, W., Miller, W., Myers, E. & Lipman, D.J.), 1990
- Three primary questions: How to select **L**? How to extend the seed? How to confirm that the match is biologically relevant?

## SELECTING SEED SIZE L

- Smaller **L**: easier to find, but decreased performance, and, importantly, *specificity* – two random sequences are more likely to have a short common substring
- Larger **L**: could miss out many potential matches, leading to decreased sensitivity.
- By default BLAST uses **L** (*w*, word size) of 3 for protein sequences and 11 for nucleotide sequences.
- MEGABLAST (a faster version of BLAST for similar sequences) uses longer seeds.

## HOW TO EXTEND THE MATCH?

- Simple (gapless) extension (original BLAST)

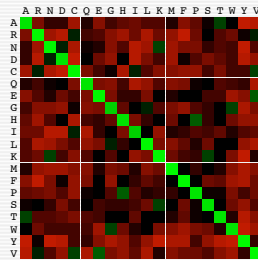


- Gapped local alignment (blastn)



## HOW TO SCORE MATCHES?

- Biological sequences are not random
  - some letters are more frequent than others (e.g. in HIV-1 40% of the genome is A)
  - some mismatches are more common than others in homologous sequences (e.g. due to selection, chemical properties of the residues etc), and should be weighed differently.
- BLAST introduces a weighting function on residues:  $\delta(i,j)$  which assigns a score to a pair of residues.
  - For nucleotides it is 5 for  $i=j$  and -4 otherwise.
  - For proteins it is based on a large training dataset of homologous sequences (**P**oint **A**ccepted **M**utations matrices). PAM120 is roughly equivalent to substitutions accumulated over 120 million years of evolution in an average protein



HIV-WITHIN

## STATISTICS OF SCORES

- Given a segment pair  $H$  between two sequences, comprised of  $r$ -character substrings  $T_1$  and  $T_2$ , we compute the score of the  $H$  as:

$$s(H) = \sum_{i=1}^r \delta(T_1[i], T_2[i])$$

- We are interested in finding out how likely the **maximal** score for **any** segment pair of two random sequences is to exceed some threshold  $X$

## HOW TO COMPUTE SIGNIFICANCE?

- Before a search is done we need to decide what a good cutoff value  $H$  for a match is.
- It is determined by computing the probability that two random sequences will have at least one match scoring  $H$  or greater.
- Uses Altschul-Dembo-Karlin statistics (1990-1991)

## STATISTICS OF SCORES (CONT'D)

- Dembo and Karlin (1990) showed that the expected # of High Scoring pairs, exceeding the threshold  $S'$  is

$$E' = K m n e^{-\lambda S'}$$

- $K$  and  $\lambda$  are expressions that depend on the scoring matrix and letter frequencies.
- $E$ -value is the number of sequences yielding a similarity score of at least  $S$  expected by chance
- Practical