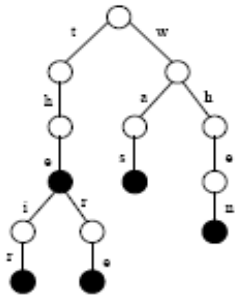
Tries and Trees

There are several interesting data structures for speeding up exact pattern matches in strings..

A *trie* is a data structure which permits access to any string s in an n word dictionary in $O(|s|)$ time for constant-sized alphabets.

This is optimal and independent of the dictionary size!

A trie has a node for each character position, with prefixes shared:



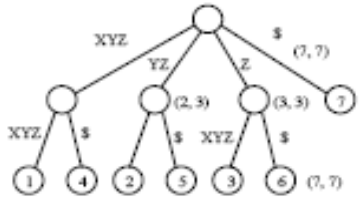
Searching in a trie is easy: just match the character and traverse down the correct path.

Building the trie is also easy: insert a new string by matching until you fail, then split the last node.

Suffix Trees

A very special set of patterns to put in a trie are the complete set of all *suffixes* of a string.

```
XYZXYZ$
 YZXYZ$
  ZXYZ$
   XYZ$
    YZ$
     Z$
      $
```



With such a tree, we can perform *substring* searches efficiently, since every substring is the prefix of some suffix.

Further, the set of all instances of a given substring t are the leaves of the subtree rooted at t, and can be found by DFS.

Suffix Trees

A suffix tree can be stored in linear space, by collapsing degree-1 nodes into paths, and paths into references to the original string.

The incremental insertion algorithm to build a suffix tree might take $O(n^2)$ time to build the tree, because finding the split-point for each insertion might require $O(n)$ time in matching.

However, there are more sophisticated algorithms (Weiner's, Ukkonen's, McCreight's) which can build the *entire tree* in linear time.

Solving Longest Common Substring with suffix trees

Given two strings s_1 and s_2, what is the longest *contiguous* substrings they have in common.

Example: *live*stock and sea*live*r

The naive $O(nm)$ algorithm fully tests each alignment of s_1 against s_2.

In 1970, Knuth conjectured that a linear-time algorithm was impossible. Can you prove him wrong?

Build a suffix tree of the length $n+m+1$ concatenated string $s' = s_1 \# s_2$, where $\#$ does not occur in either string.

Label each leaf node of the suffix tree with the name of the string it is contained in. Label each internal node with the union of the labels of its descendents.

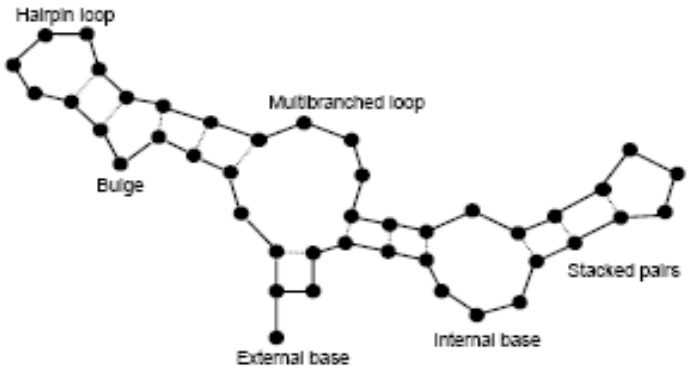
By doing a DFS on the $O(n+m+1)$ node tree, we can find the deepest node which has both an s_1 and s_2 descendant. This defines the longest common substring!

Solving Palindromes with suffix trees

A *palindrome* is a string which reads the same forwards and backwards, e.g. *A man, a plan, a canal – Panama* or *Madam I'm Adam*.

In DNA sequence analysis, a palindrome is a sequence equal to its reverse complement, e.g. *GAATTC*.

Such palindromes bind/fold to create *secondary structures* in sequences, which often have biological significance.



Solving Palindromes with suffix trees

How can we find the longest self-binding substring in a DNA sequence? Use the longest common substring algorithm with s_1 as the input sequence and s_2 as its reverse complement sequence!

This does not guarantee that the lcs of these strings starts/ends in the same place, so does not necessarily find a palindrome.

However, after we augment the suffix tree so as to answer *lowest common ancestor* queries in constant time...

...we can find the longest sub-palindrome in linear time by asking the 'length' of the LCA of $S[i]$ and $S[i+n+1]$ for each $1 \leq i \leq n$.

Solving Circular String Linearization with suffix trees

Certain genomic structures (*plasmids*) have closed circular DNA molecules rather than linear molecules.

To look such a molecule up in a database, we much find a *canonical* place to break it to leave a linear string.

The most obvious place to break it uniquely is so as to always leave the *lexicographically* smallest string, i.e. the string which appears first in sorted order.

Building and sorting all n such strings takes $O(n^2)$ time. Can you do better?

Break the string arbitrarily to create a linear string L.

Now build the suffix tree for string $S = LL\#$. This is linear in the size of the input.

Example: $L = gcttcaat$ so $S = gcttcaatgcttcaat\#$.

Do a traversal down from the root, always picking the lexicographically smallest character. Assume that # is at the end of the alphabet.

Suffix Arrays

The *suffix array* is an amazing data structure for efficiently searching whether S is a substring of string T.

For a given string T, we construct the lexicographically sorted array of all its *suffixes*.

For $T = mississippi$, the suffix array is:

```
11 : i
 8 : ippi
 5 : issippi
 2 : ississippi
 1 : mississippi
10 : pi
 9 : ppi
 7 : sippi
 4 : sissippi
 6 : ssippi
 3 : ssissippi
```

Since every substring is the prefix of some suffix, Substring search now reduces to binary search in this array. Example: is "sip" a substring of T?

Once you have the suffix array, the search time is $O(\lg n + m)$, where n is the length of T and m the length of the matched substring.

Note that we can just as easily find *all* the occurrences of a given string S in T by binary searching just before/after S.

Building and storing suffix arrays

The really amazing thing is that one need only store the original string and the sorted start positions to do the search! The jth character of the ith prefix is at $T[start[i] + j - 1]$.

But how fast can be built the suffix array of an n character string?

Radix sorting n strings of n characters can be done in $O(n^2)$, linear in the size of the input.

But what is really amazing is that suffix arrays can be built in both linear time and space!

First, build a *suffix tree* in $O(n)$ time.

Performing a lexicographic depth first search of a suffix tree yields a suffix array.

Suffix arrays use many times less space than suffix trees (say $3n$ vs. $17n$ bytes), which is often the dominating factor in large text search problems.

Constructing the overlap graph

Through clever use of suffix arrays, the entire overlap graph can be built in near-linear time.

After building the array of all suffixes of all fragments and their *reverses*, potential overlaps will share a prefix of a suffix, and hence be near each other in sorted order.

Accepting a fragment pair as overlapping may require several significant long matches.

Since there are 4^k possible DNA sequences of length k, and n places for such k-mers to start if $|T| = n$, matches start to get significant if $k > \log_4 n$.

For human, longer than 16-mers start to get interesting, so we can expect to find significant exact matches.

Complications

Several engineering issues arise in building any assembler:

- *Chimera detection* – Certain fragments are, in fact concatenations of sequences from two different regions.
- *Multiple sequence alignment* – We can remove single base sequencing errors by voting in overlapped regions.
- *Identifying repeat regions* – Mapping data and ad hoc algorithms are essential to help resolve repeats.
- *Integrating distance and clone constraints into assembly* – This requires careful laboratory work (preferably automated) to ensure accurate input.

Gel reading programs

Calling sequence bases from sequence trace data is not a trivial problem for several reasons, including (1) varying density of the gel along a lane, (2) varying density of the gel across lanes, (3) separation between bands shrinks as we move along the lane.

Phred, by Phil Green, is the most famous base-calling program for automated sequencer traces. It assigns an error probability to each base called and yielded 40-50% lower errors than the software included with Applied Biosystems (ABI) sequencing machines.

Gel reading programs

Phred's base-calling pipeline consists of several phases:

1. It predicts peak locations using the assumption that fragments should be locally evenly spaced. This helps determine the correct number of bases in a region where peaks are not well resolved, noisy or displaced.

2. Observed peaks are identified in the trace and matched to the predicted peak locations. Some are omitted and some are split, yielding the main base sequence.

3. Unmatched peaks are analyzed to see if they represent bases, and if so is inserted into the sequence.

The quality score Q is based on an estimate of the probability of error p, where $Q = -10 \log_{10} p$.

Assemblers: The first generation

These were designed to assembly small (contig-size) sequencing projects, say 40 kb to 100 kb or so. Thus quadratic time/space algorithms potentially suffice.

CAP, by Xiaoqiu Huang, is a representative contig assembly program. It does a dynamic programming alignment between pairs of fragments, taking into account that errors increase at the beginning and the end of the read.

Assemblers: The first generation

Phrap was designed to work with Phred, and exploits base quality scores if they exist. Important processing steps include:

1. Masking likely 'garbage' sequence at the beginning and end of reads. Phrap looks for regions consisting almost entirely of a single base; such regions are likely due to poor data quality.

2. Identifying all potentially overlapping pairs of sequences. Two overlapping sequences must have at least one exact match of length *minmatch* (typically 14 bases), and the alignment between the sequences must have a score of at least *minscore* (default: 30).

3. Modified quality scores are calculated for each base in each read, taking into account confirmation by overlapping reads as well as orientation and sequencing chemistry. "LLR" scores are computed for each pairwise alignment, measuring of overlap length and quality. High quality discrepancies that potentially indicate different copies of a repeat lead to low LLR scores. Potential problem clones like chimeras are also identified.

4. Merge reads into contigs, starting at the pairwise overlaps with the highest LLR scores. Likely chimeras are not used.

5. A consensus sequence is extracted from the merged sequence, based on voting from the highest adjusted quality scores at any base.

Assemblers: The new generation

Assemblers for bacterial-sized genomes and beyond (such as those used by TIGR and Celera) must use more sophisticated data structures to avoid comparing every pair of fragments.

Typically, an initial assembly of 'unitigs' of high-quality, long overlaps is constructed.

The expected number of fragments at every position is a function of coverage. Stacks of fragments which are too high likely indicate improperly combined repeat regions.

These unitigs are assembled into a 'scaffold' of pieces. Most reads are in 'mate pairs', since the left and right ends of clones of sizes up to 10 kb or so have both been sequenced. The scaffold is assembled using this pairing and distance information.

Other less high-quality reads and unitigs can now be positioned on this scaffold. Remaining gaps can be closed by sequencing the clones spanning different contigs of the scaffold.

Sequencing tricks

Celera used the public consortium's Human sequence data from Genbank. To avoid being fooled by its assembly, they computationally shredded this sequence into artificial reads and incorporated this as raw data into its own project.

Certain sequencing projects do not attempt to sequence full genomic DNA. To identify the genes, they translate back the *RNA* transcripts of genes into *cDNA* and sequence this.

Although the genes are a very important part of what we are looking for, a drawback such expressed sequence tags (ESTs) is that more highly expressed genes are significantly over-represented, making it hard to find rare genes.

Sequencing a mixed population (i.e. fragments from multiple individuals of a given species) should not compromise assembly because of the overwhelming in-species similarity, while differences in bases helps study variation (single nucleotide polymorphisms or SNPs)