Lecture Notes on
Data Structures

M1522.000900

Seoul National University

Fall 2022

Part IX

String Search

String Search

Applications:

- Data: strings such as English dictionary, DNA sequences.
- Queries:
  - exact match (*e.g.*, "computer", "Tucson")
  - prefix match (*e.g.*, "inter*", "conc*"),
  - substring match (*e.g.*, "*connect*"),
  - subsequence match (*e.g.*, "*c*o*n*n*e*c*t*").

Trie: re<u>TRIE</u>val

- An alternative to hashing for string search.

Standard Tries

Assumption:

- $\nexists(x, y)$ such that x is a prefix of y.
- What if this condition is not satisfied?

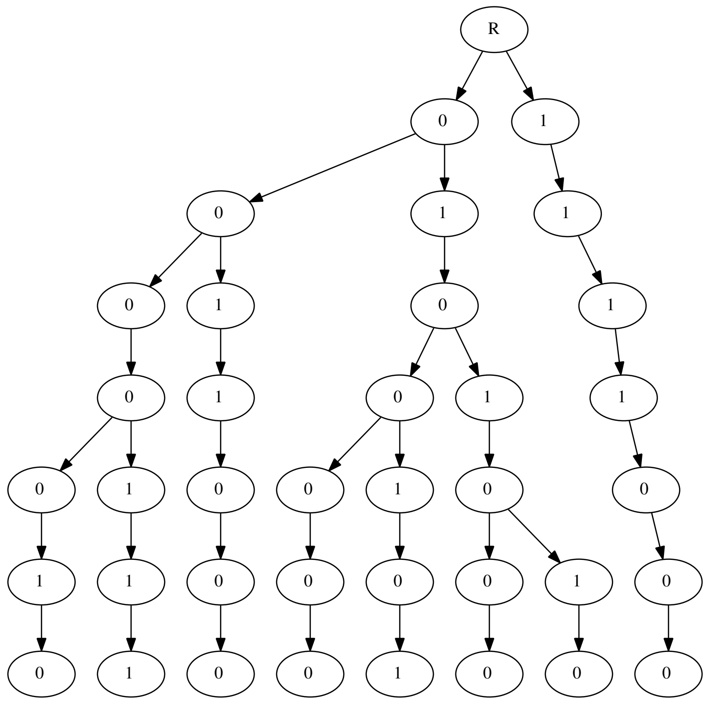
Structure:

- The max number of children a node can have is the size of alphabet. (*e.g.*) 2 for binary strings, 4 for DNA sequences.
- Each node stores one character. (*e.g.*) a bit for binary strings, one of A/C/G/T for DNA sequences.
- A dummy root node is necessary. (A trie with a dummy root only is considered empty.)
- The depth of a trie is one plus the max length of strings.
- There is a 1-to-1 correspondence between leaf nodes and strings.

Example 1

Build a trie for the following 7-bit binary numbers.

0000010 0100101 0000111 0101000
0011000 0101010 0100000 1111000



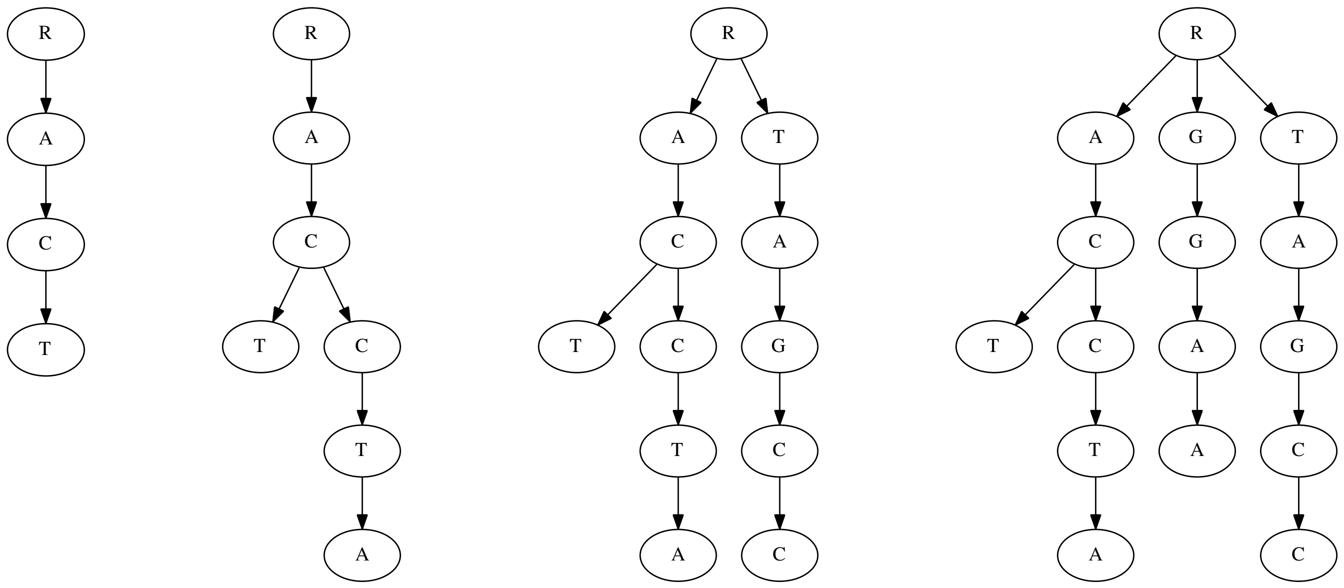
Trie: Insertion

1. Traverse down using the i^{th} character of a new string to make a branching decision at the i^{th} level of the trie.
2. Stop if no matching branch is found for the character (*i.e.*, the new string is not in the trie).
3. Create a new chain of nodes to store the remaining characters.
   - (Ex) Insert a new binary string 1011111 into the trie in Example 1.

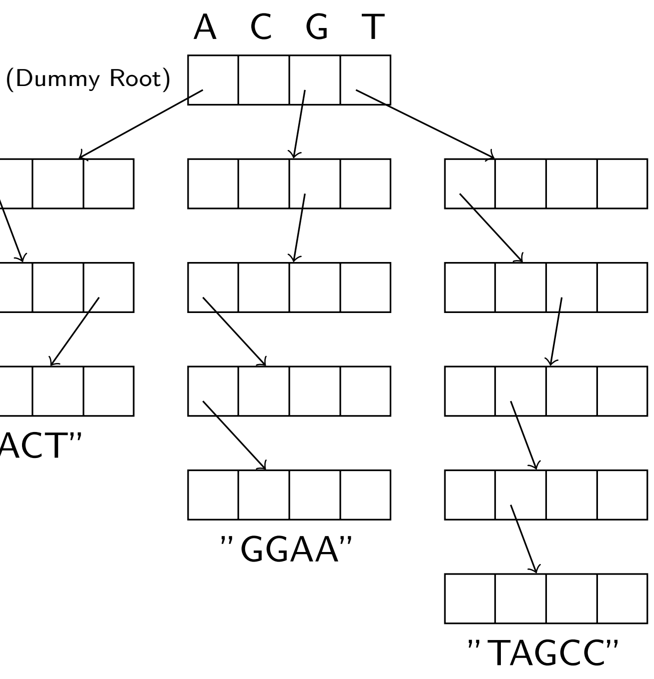
Example 2

Insert the following DNA sequences into an initially empty trie.

`ACT, ACCTA, TAGCC, GGAA.`



Implementation of Trie



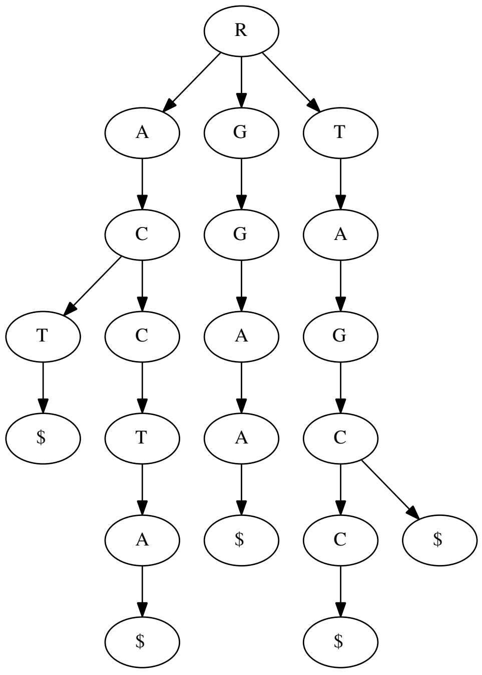
A requirement for Trie

- Can we add `TAGC` to the trie built in Example 2?
- If a key x is a prefix of another key y, a trie cannot store both the keys at the same time.
- To ensure that each key is uniquely associated with a leaf node, attach a special character not in the original alphabet at the end of each key string.

Example 3

Build a trie for the following DNA sequences.

`ACT$, ACCTA$, TAGCC$, GGAA$, TAGC$`



Trie: Search

Use the characters in the search string to traverse the trie.

- An *exact-match* search succeeds only if it terminates at a leaf node (after using up all characters in the search string).
  - For example, in Example 3, search for `ACT$` succeeds, but search for `AC$`, `ACG$` or `ACTT$` fails.
- A *prefix-match* search succeeds if it terminates at a leaf or an internal node. All the leaf nodes in the subtree (rooted at the node the search stops at) are considered matches.
  - For example, in Example 3, a prefix search `AC*` stops at an internal node and returns `ACT$` and `ACCTA$`.
- If a leaf node is reached before using up all characters in the search string, it fails for both exact-match and prefix-match searches.

Trie: Analysis

- The number of nodes in a trie is $\mathcal{O}(\sum_i |string_i|)$.
- The cost of an insertion is $\mathcal{O}(m \times |string|)$, where m is the size of alphabet.
  - At most m comparisons are made at each node.
  - The m factor can be removed if *direct addressing* is feasible.
- The cost of a search is $\mathcal{O}(m \times |string|)$ too.

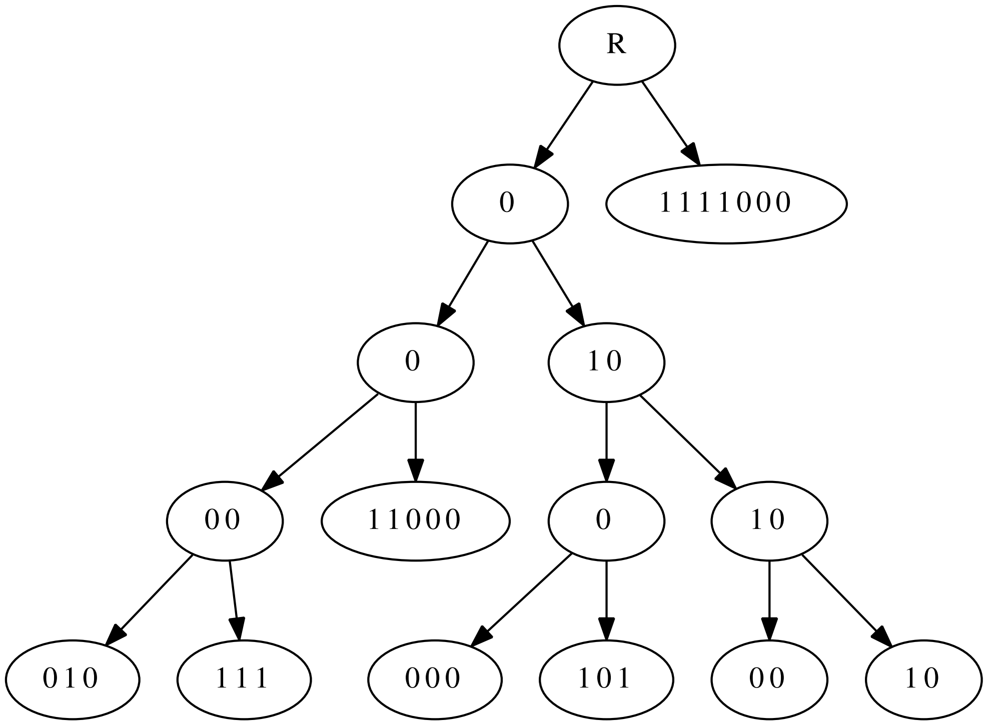
PATRICIA: Compressed Trie [Morrison]

PATRICIA: Practical Algorithm To Retrieve Information Coded In Alphanumeric

- Merge a node with its child node if the node has only one child.
  - Trie: The number of nodes is no more than the aggregate number of all characters in all strings.
  - Patricia: The number of nodes is $\mathcal{O}$(number of leaf nodes) because there is no node with only one child. (*e.g.*, a binary PATRICIA is a full binary tree. See Example 4.) The number of leaf nodes is always equal to the number of strings.
    $\therefore$ The number of nodes is $\mathcal{O}$(number of strings).
- Compact representation of a node: $[string\_id, begin, end]$
  - This allows a fixed-length record for both internal and leaf nodes.
  - But the input strings must be kept separately somewhere.
- The space required for a Patricia with n strings is $\mathcal{O}(m \times n)$ where m is the size of alphabet. Usually, $n \gg m$.

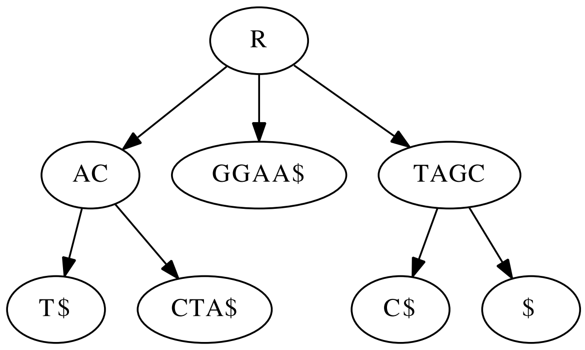
Example 4

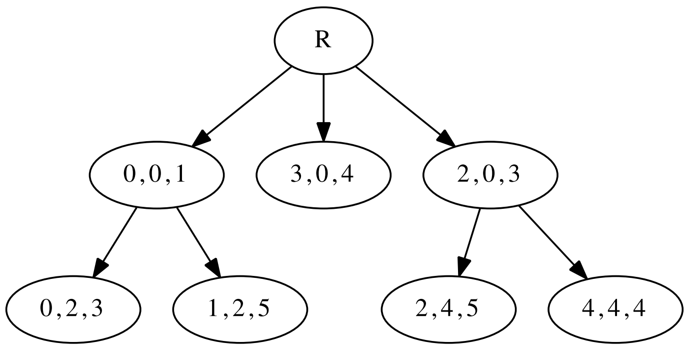
Compress the trie built in Example 1.



Example 5

Compress the trie built in Example 3 using the compact representation.





str id	offset					
	0	1	2	3	4	5
0	A	C	T	$		
1	A	C	C	T	A	$
2	T	A	G	C	C	$
3	G	G	A	A	$	
4	T	A	G	C	$	

Problem 1

How much space is saved by compressing a trie that stores n DNA sequences?

Let M be $\sum_i |sequence_i|$. Recall $m = 4$ for DNA sequences.

A standard trie stores none of the DNA sequences separately but will have as many nodes as $\mathcal{O}(M)$. It thus requires $\mathcal{O}(4 \times M)$ memory just for the trie.

A compressed trie will have $\mathcal{O}(n)$ nodes in addition to all the DNA sequences stored separately. Thus, the compressed trie will use $\mathcal{O}(M + (4 + 3) \times n)$ memory for both the DNA sequences and the compressed trie.

Substring match

Substring match problem:

- For a given string X and a search pattern P, determine whether P is a substring of X.
- (*e.g.*) Does a fragment, "*ATC*", appear in a DNA sequence?
- A brute-force substring match takes $\mathcal{O}(|P| \times |X|)$ time.
- A trie (or Patricia) cannot handle substring queries.

Suffix Trie

Create all suffix strings of a given string X. Then, build a trie for the suffix strings.

- Append a special character to each suffix string.
- Time to build: $\mathcal{O}(m \times |X|^2)$.
- A more complex $\mathcal{O}(m \times |X|)$ algorithm exists.
- The number of leaf nodes is equal to the number of suffixes, which is equal to the length of X. $\therefore$ The space requirement is $\mathcal{O}(|X|)$ by the compact representation.

P is a substring of X iff there exists a path from the root to node v and P is a *prefix* of the string of the *root* $\rightsquigarrow$ v path.

- Time to search: $\mathcal{O}(m \times |P|)$.
- The first match is found but not all occurrences of P.

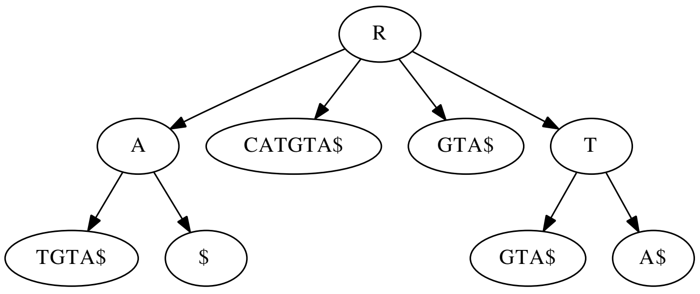
Example 6

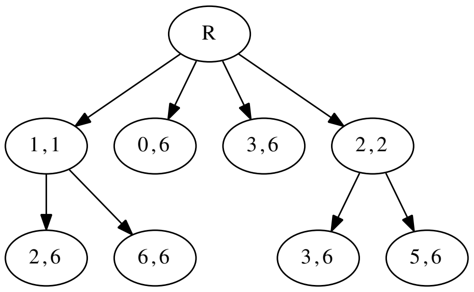
Are fragments `GTA` and `ATGT` a substring of the following DNA sequence?

`CTTACCGCTTAAGCCTAGCTAGTACATGTA.`

Suffix strings:

```
     A$
    TA$
   GTA$
  TGTA$
 ATGTA$
CATGTA$
    ...
```





Substrings vs. Subsequences

Definition 1

A subsequence of string $X = x_1x_2\cdots x_m$ is any string of the form $x_{i_1}x_{i_2}\cdots x_{i_k}$, where $i_j < i_{j+1}$. That is, a subsequence of X is a sequence of characters, not necessarily contiguous, but taken in order from X.

- Subsequences of $X = ABCBDAB$ are

$$A, B, C, D, AB, BB, AC, ACB, BBB, BCBD, \ldots$$

- A substring and a subsequence are different.
- The number of non-null substrings of X is $|X|(|X|+1)/2$.
- The number of non-null subsequences of X is $2^{|X|} - 1$.

Trie for Subsequence Queries?

Problem 2

You can still use a trie to answer subsequence match queries by storing all possible subsequences of a string, say X, in the trie.

1. How can you process a subsequence query, say Q, with the trie?
2. What is the disadvantage of this approach?

1. Run an exact match query with the trie. It takes $\mathcal{O}(|Q|)$ time.
2. The size of the trie will grow too fast, that is, exponentially.

Problem 3

How long does it take to determine whether Q is a subsequence of X (without using a trie or any other data structure)?

It takes $\mathcal{O}(|X|)$ time. Slow if $|X| \gg |Q|$.

Longest Common Subsequence (LCS)

Consider DNA strands from two different individuals. They may be considered genetically related if they have a long subsequence common in their DNA sequences.

- For two strings $X = ABCBDAB$ and $Y = BDCABA$, common subsequences are $A, AB, BC, BDA, BDB, BDAB, \ldots$

Formally, the LCS problem is stated as follows.

Definition 2

Given two strings $X = x_1x_2\cdots x_m$ and $Y = y_1y_2\cdots y_n$, find a longest string that is a subsequence of X and Y.

- A brute-force approach will take $\mathcal{O}(2^{|X|} \times |Y|)$ time, because the number of subsequences of X is $\mathcal{O}(2^{|X|})$ and it takes $\mathcal{O}(|Y|)$ time (refer to Problem 3) to determine if each of X subsequences is a subsequence of Y.

We can do better than that by *Dynamic Programming*.

The idea is

1. If $x_m = y_n$, then $LCS(X, Y) = LCS(X_{m-1}, Y_{n-1}) \cdot x_m$.
2. Otherwise, $LCS(X, Y)$ is either $LCS(X_{m-1}, Y)$ or $LCS(X, Y_{n-1})$. Whichever is longer is an $LCS(X, Y)$.

Notations: X_i is a prefix string of X, *i.e.*, $X_i = x_1 x_2 \cdots x_i$ for $i \le m$. X_0 is an empty string.

For example, LCS(`ABCBDAB`, `BDCABA`) is either LCS(`ABCBDA`, `BDCABA`) or LCS(`ABCBDAB`, `BDCAB`), because the last characters of the two strings are different.

Algorithm 1 (LCS-Length)

```
for(i=0; i <= m ;i++) L[i,0] = 0;
for(j=0; j <= n ;j++) L[0,j] = 0;
for(i=1; i <= m ;i++)
   for(j=1; j <= n ;j++)
      if ($x_i$ == $y_j$) L[i,j] = 1 + L[i-1,j-1];
      else L[i,j] = max(L[i-1,j], L[i,j-1]);
```

For two strings $X = x_1 x_2 \cdots x_m$ and $Y = y_1 y_2 \cdots y_n$, a two-dimensional $(m+1) \times (n+1)$ matrix stores $|LCS(X_i, Y_j)|$ of all pairs of prefix strings X_i and Y_j ($1 \le i \le m$ and $1 \le j \le n$). $L[m, n]$ is the length of LCS(X,Y).

$$L[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ L[i-1, j-1] + 1 & \text{if } x_i = y_j, \\ max\{L[i, j-1], L[i-1, j]\} & \text{if } x_i \neq y_j. \end{cases}$$

Example 7

How long is the longest common subsequence of `BDCABA` and `ABCBDAB`?

	0	1 (A)	2 (B)	3 (C)	4 (B)	5 (D)	6 (A)	7 (B)
0	0	0	0	0	0	0	0	0
1 (B)	0	0	1	1	1	1	1	1
2 (D)	0	0	1	1	1	2	2	2
3 (C)	0	0	1	2	2	2	2	2
4 (A)	0	1	1	2	2	2	3	3
5 (B)	0	1	2	2	3	3	3	4
6 (A)	0	1	2	2	3	3	4	**4**

- If $X_i = Y_j$, copy diagonally and increment by one.
- If $X_i \neq Y_j$, copy the larger or the top if equal.

Problem 4 (LCS-Print)

Modify Algorithm 1 so that it constructs an LCS of X and Y.

```
...
if (x_i == y_j) { L[i,j] = 1 + L[i-1,j-1]; p[i,j]='d'; }
if (L[i-1,j] ≥ L[i,j-1]) { L[i,j] = L[i-1,j]; p[i,j]='t'; }
else { L[i,j] = L[i,j-1]; p[i,j]='l'; }
```

Starting from $p[m,n]$, follow the path directed by 'd' (diagonal), 't' (top) and 'l' (left), and print x_i (or y_j) if $p[i,j] = d$. Then, reverse the printed string.

The algorithm given in Problem 4, starting from `L[6,7]`, returns just one LCS instance, `BDAB`. Is this the only LCS?

Problem 5

Find all LCS instances of `BDCABA` and `ABCBDAB`.

The fact $\mathtt{L}[5,7] = \mathtt{L}[6,7] = 4$ indicates that `LCS(BDCAB,ABCBDAB)` is also an LCS of the two input strings. Backtracking from `L[5,7]` will return another LCS, `BCAB`. The other LCS is `BCBA` (or $x_2x_3x_4x_6$ and $y_1y_3y_5y_6$).

Problem 6

Write an algorithm that prints all LCS instances of X and Y.