Bad Character Rule (BCR) (3)

- On a mismatch, shift the pattern to the right until the first occurrence of the mismatched character in P.

- Still O(n m) worst case running time:

```
T: aaaaaaaaaaaaaaaaaaaaaaaaa
P: abaaaa
```

Good Suffix Rule (GSR) (1)

- We want to use the knowledge of the matched characters in the pattern's suffix.
- If we matched S characters in T, what is (if exists) the smallest shift in P that will align a sub-string of P of the same S characters ?

Good Suffix Rule (GSR) (2)

- **Example 1 – how much to move:**

```
T: bbacdcbaabcddcdaddaaabcbcb
P: cabbabdbab
         cabbabdbab
```

Good Suffix Rule (GSR) (3)

- **Example 2 – what if there is no alignment:**

```
                                       ↓
T:  bbacdcbaabcbbabdbabcaabcbcb
P:  bcbbabdbabc
             bcbbabdbabc
```

Good Suffix Rule (GSR) (4)

- We mark the matched sub-string in T with *t* and the mismatched char with *x*
  1. In case of a mismatch: shift right until the first occurrence of *t* in P such that the next char *y* in P holds $y \neq x$
  2. Otherwise, shift right to the largest prefix of P that aligns with a suffix of *t*.

Good Suffix Rule (GSR) (5)

- L(i) – The biggest index j, such that j < n and prefix P[1..j] contains suffix P[i..n] as a suffix but not suffix P[i-1..n]

i	1	2	3	4	5	6	7	8	9
P	G	T	A	G	C	G	G	C	G
L(i)	**0**	**0**	**0**	**0**	**0**	**0**	**6**	**0**	**7**

Good Suffix Rule (GSR) (6)

- l(i) – The length of the longest suffix of P[i..n] that is also a prefix of P

i	1	2	3	4	5	6	7	8	9
P	G	T	A	G	C	G	G	C	G
l(i)	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Good Suffix Rule (GSR) (7)

- Putting it together
  - If mismatch occurs at position n, shift P by 1
  - If a mismatch occurs at position i-1 in P:
    - If L(i) > 0, shift P by n – L(i)
    - else shift P by n – l(i)
  - If P was found, shift P by n – l(2)

Boyer Moore Algorithm

- Preprocess(P)
- k := n

while (k $\leq$ m) do

  - Match P and T from right to left starting at k
  - If a mismatch occurs: shift P right (advance k) by max(good suffix rule, bad char rule).
  - else, print the occurrence and shift P right (advance k) by the good suffix rule.

Boyer-Moore Algorithm Demo

GTTATAGCTGATCGCGGCGTAGCGGCGAA

GTAGCGGCG

Bad Character Rule – shift by 7
Good Suffix Rule – shift by 0

Boyer-Moore Algorithm Demo

GTTATAGCTGATCGCGGCGTAGCGGCGAA

GTAGCGGCG *t*

Bad Character Rule – shift by 1
Good Suffix Rule – shift by 3 (9-6)

Pattern P - GTAGCGGCG **t - GCG**

Prefixes of P

G
GT
GTA
GTAG
GTAGC
GTAGCG

Find the longest prefix of P which has t as the suffix.

Boyer-Moore Algorithm Demo

GTTATAGCTGATC$\boxed{\text{GCGGCG}}$TAGCGGCGAA

GTA$\boxed{\text{GCGGCG}}$ *t*

Pattern P – GTAGCGGCG

t – GCGGCG

Prefixes of P

G
GT
GTA
GTAG
GTAGC
GTAGCG

GTAGCGG
GTAGCGGC
GTAGCGGCG

Find the longest prefix of P which has *t* as the suffix.

Boyer-Moore Algorithm Demo

GTTATAGCTGATCGCGGCGTAGCGGCGAA

GTAGCGGCG *t*

Bad Character Rule – shift by 3

Good Suffix Rule – shift by 8 (9-1)

Pattern P – GTAGCGGCG *t* **– GCGGCG**

suffixes of *t*	Prefixes of P
G	G
CG	GT
GCG	GTA
GCGG	GTAG
GCGGC	GTAGC
GCGGCG	GTAGCG

Find the longest prefix of P which is a suffix of *t*.

Boyer-Moore Algorithm Demo

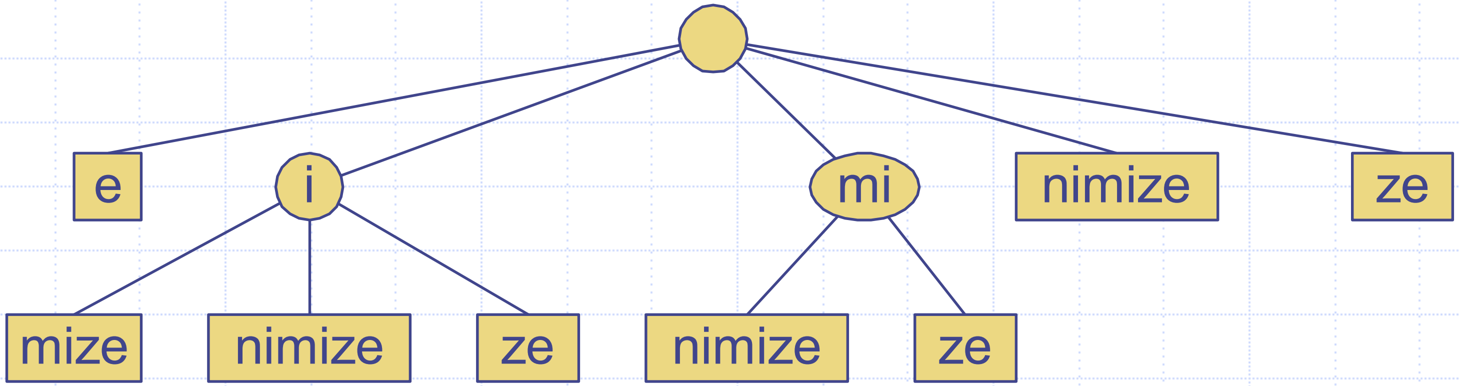
GTTATAGCTGATCGCGGCGTAGCGGCGAA

GTAGCGGCG

Boyer-Moore Algorithm Analysis

- Worst case
  - O(nm) if the pattern does occur in the text
  - For patterns of small size, the algorithm might not be efficient.
- O(n+m) if the pattern does not occur in the text

Tries



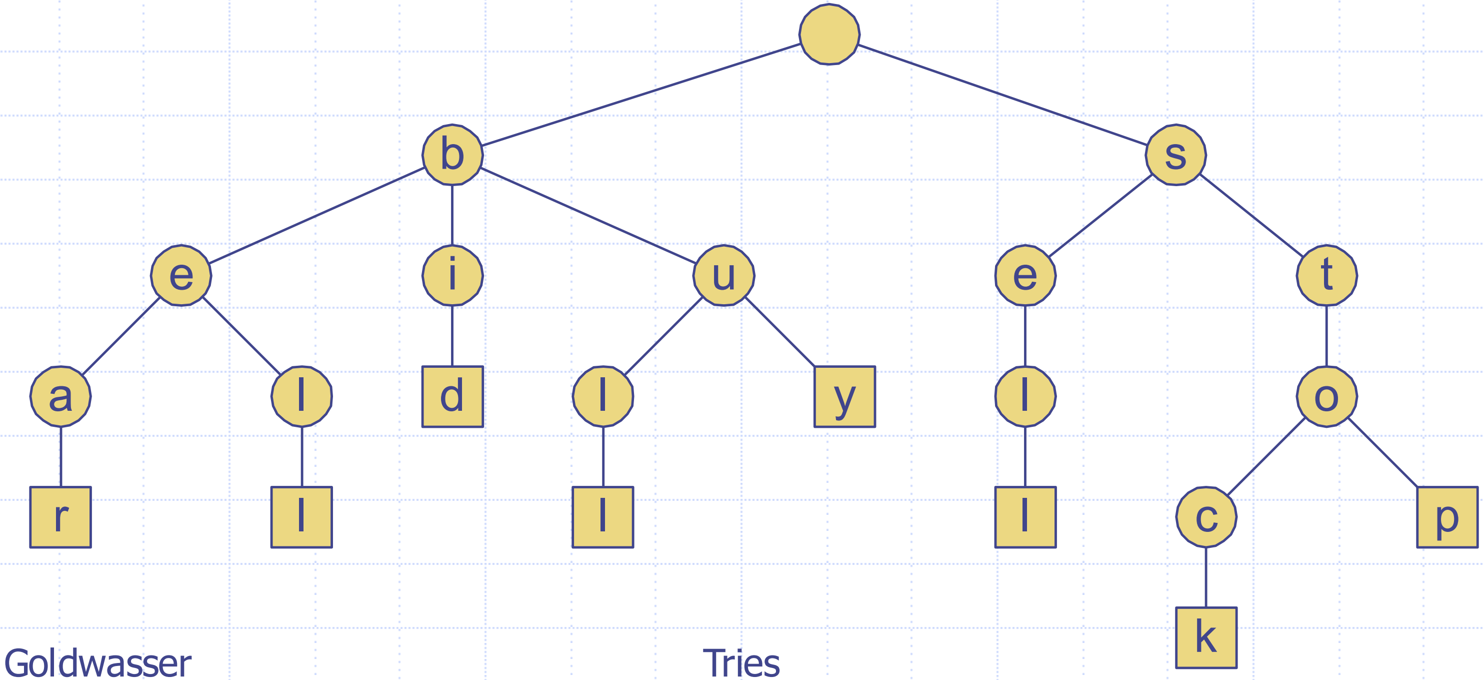
Preprocessing Strings

- Preprocessing the pattern speeds up pattern matching queries
- If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern
- A trie is a compact data structure for representing a set of strings, such as all the words in a text
  - A trie supports pattern matching queries in time proportional to the pattern size

Standard Tries

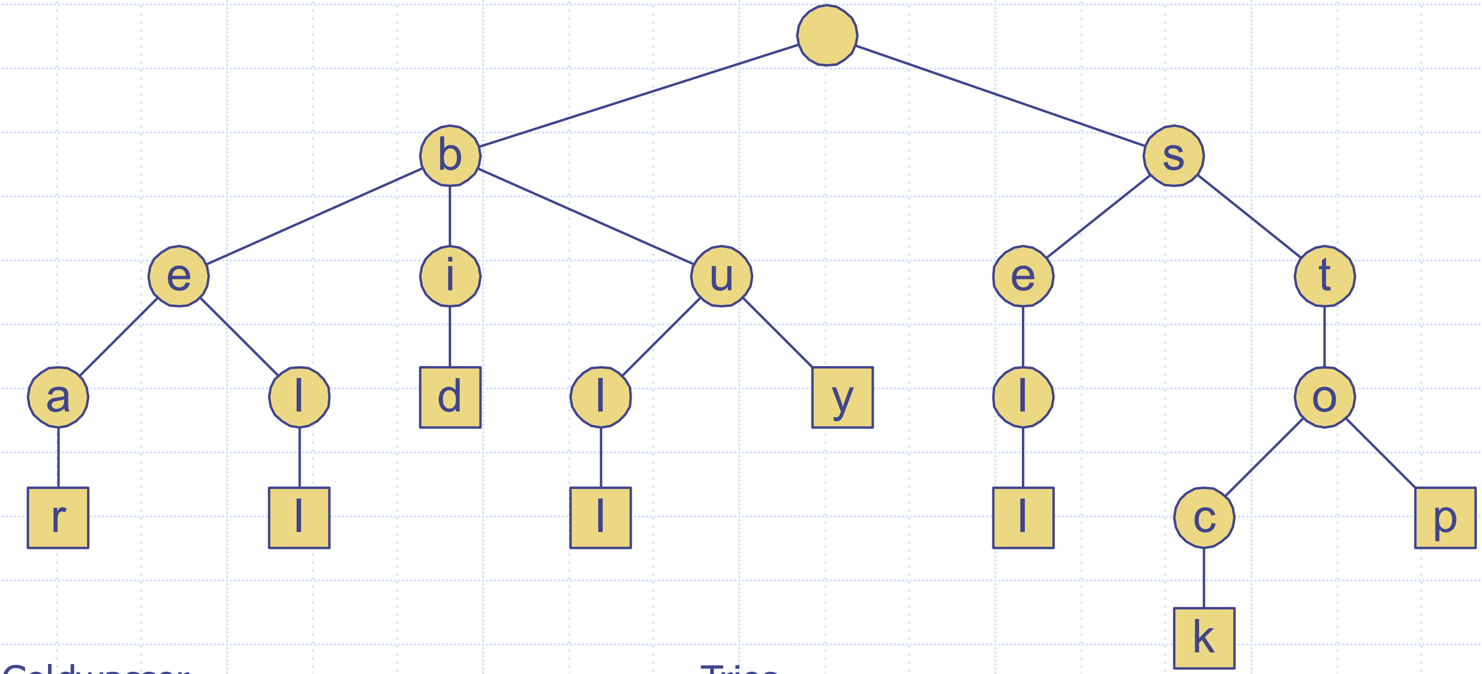
- The standard trie for a set of strings S is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered
  - The paths from the external nodes to the root yield the strings of S
- Example: standard trie for the set of strings

  S = { bear, bell, bid, bull, buy, sell, stock, stop }



Analysis of Standard Tries

- A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:
  - n total size of the strings in S
  - m size of the string parameter of the operation
  - d size of the alphabet



Application of Tries

- A standard trie supports the following operations on a processed text in O(m) time, where m is the length of the string.
  - word matching: find the first occurrence of word X in the text.
  - prefix matching: find the first occurrence of the longest prefix of word X in the text.

Word Matching with a Trie

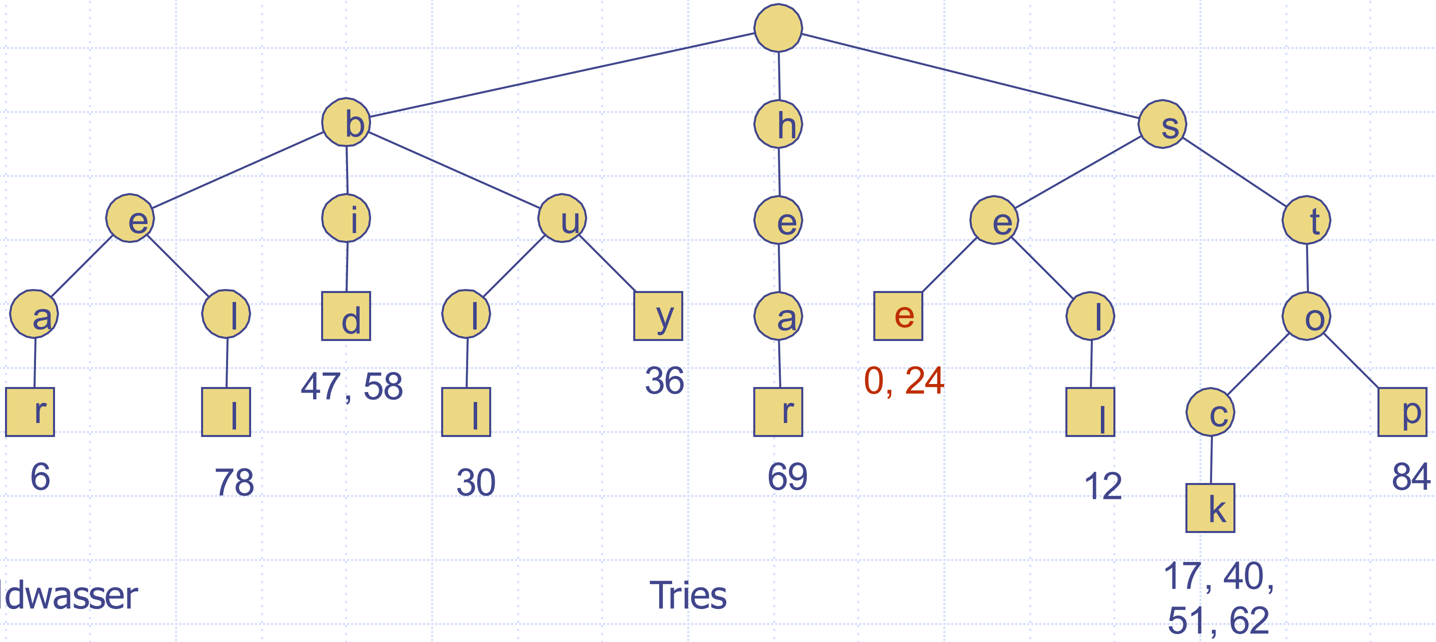
- insert the words of the text into trie
- Each leaf is associated w/ one particular word
- leaf stores indices where associated word begins ("see" starts at index 0 & 24, leaf for "see" stores those indices)

s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!	
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46

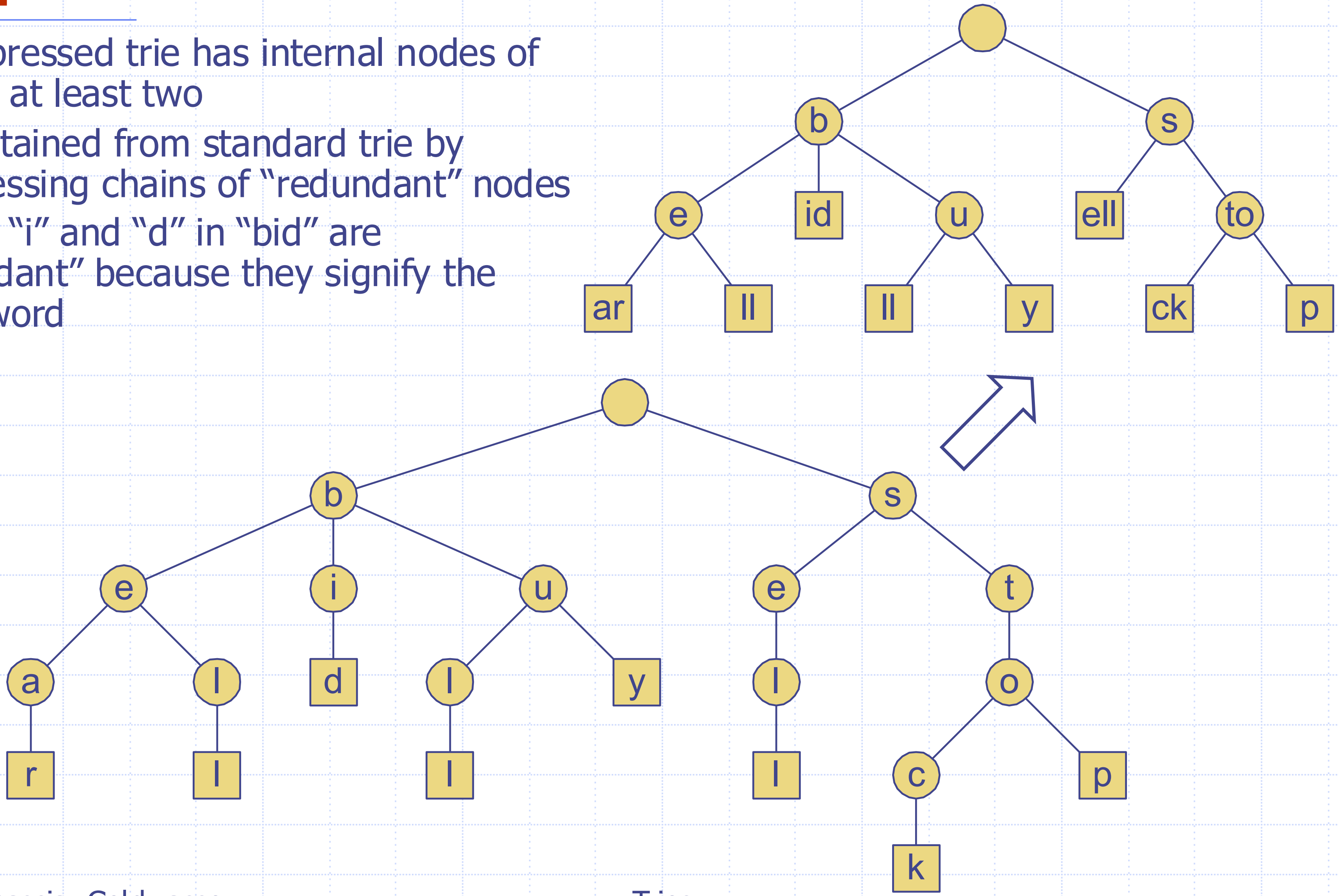
b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!	
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68

h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88



Compressed Tries

- A compressed trie has internal nodes of degree at least two
- It is obtained from standard trie by compressing chains of “redundant” nodes
- ex. the “i” and “d” in “bid” are “redundant” because they signify the same word

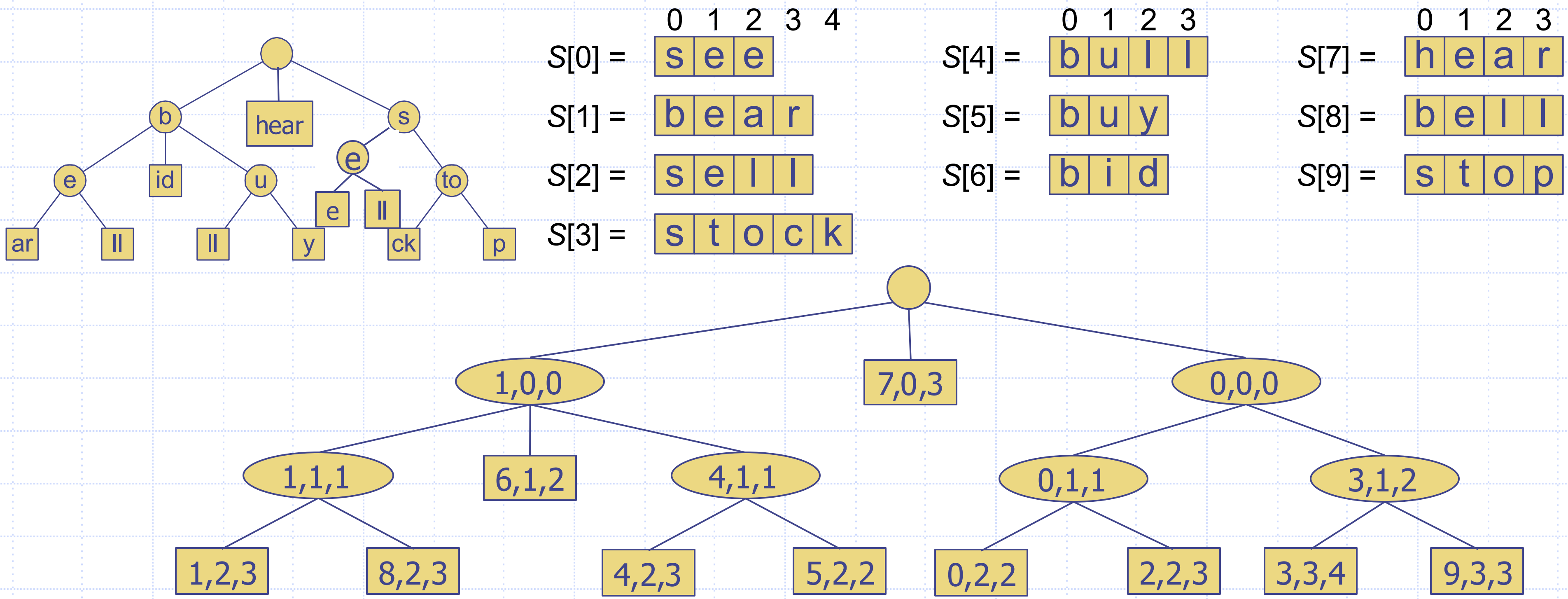


Why Compressed Tries?

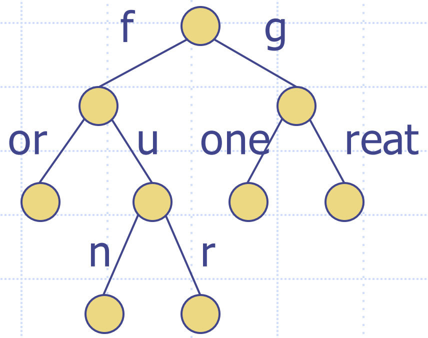
- A compressed trie storing a collection *S* of *s* strings from an alphabet of size *d* has the following properties
  - Every internal node of T has at least two children and at most d children
  - T has *s* leaf nodes
  - The number of internal nodes of T is O(s)

Compact Representation

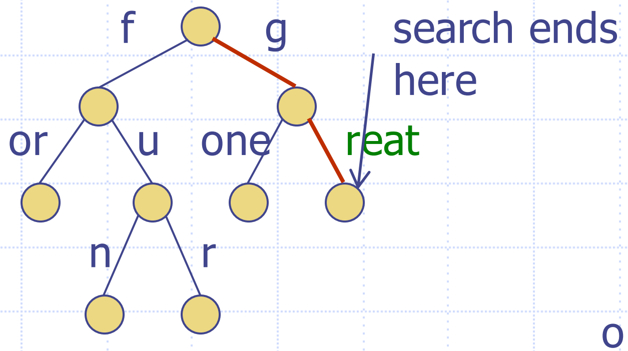
- Compact representation of a compressed trie for an array of strings:
  - Stores at the nodes ranges of indices instead of substrings
  - Uses $O(s)$ space, where s is the number of strings in the array
  - Serves as an auxiliary index structure



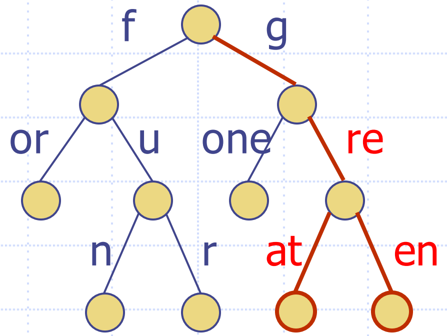
Insertion/Deletion in Compressed Trie



insert green

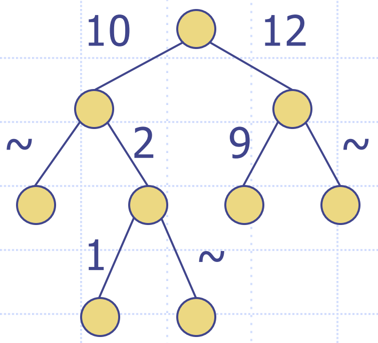


insert green



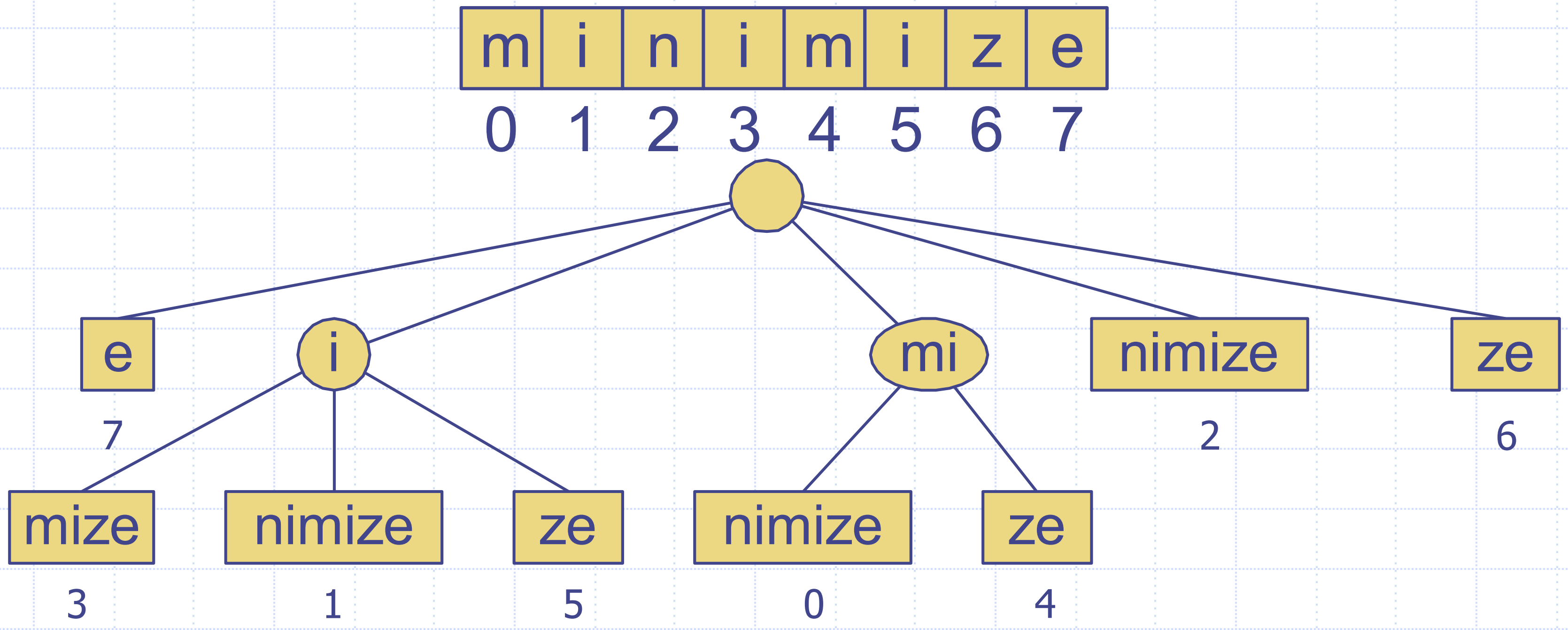
Application - Routing through Tries

- Internet Routers maintain a Trie (table)
- It is not a lookup
  - forwards packets to its neighbors using IP prefix matching rules



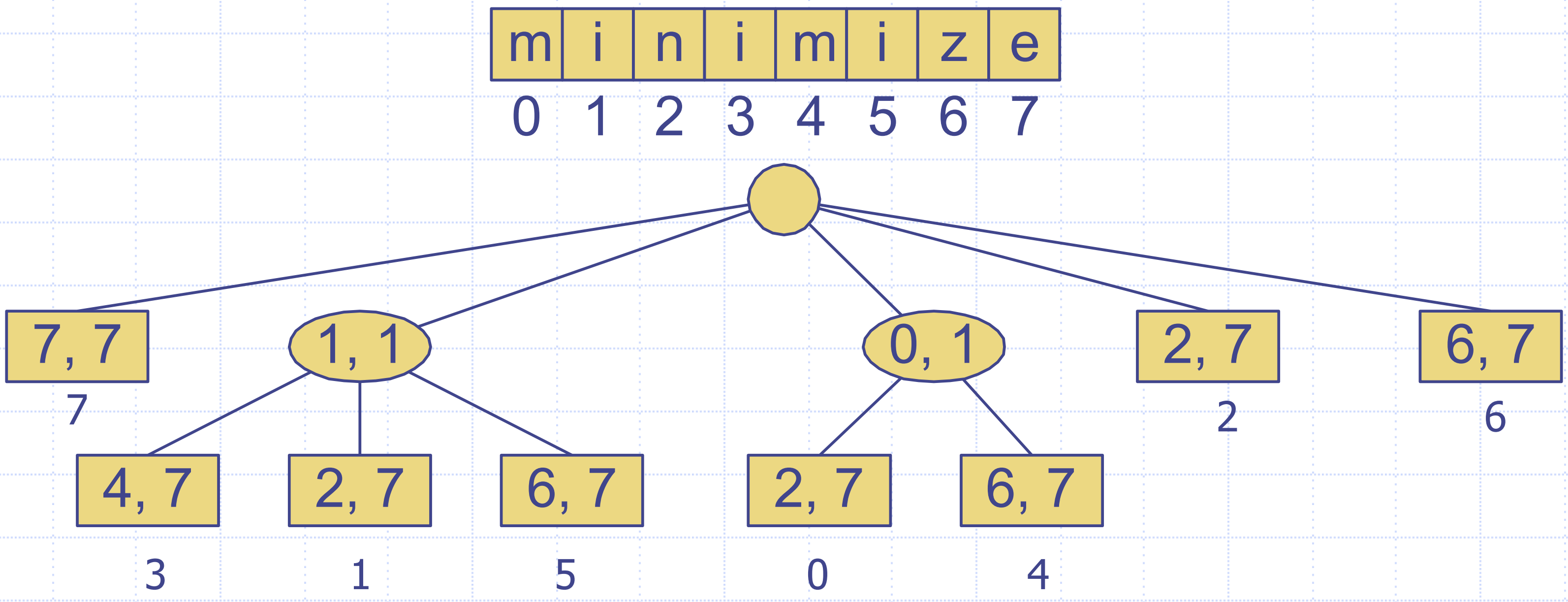
Suffix Trie

- The suffix trie of a string ***X*** is the compressed trie of all the suffixes of ***X***
- Each leaf corresponds to a suffix of ***X***



Analysis of Suffix Tries

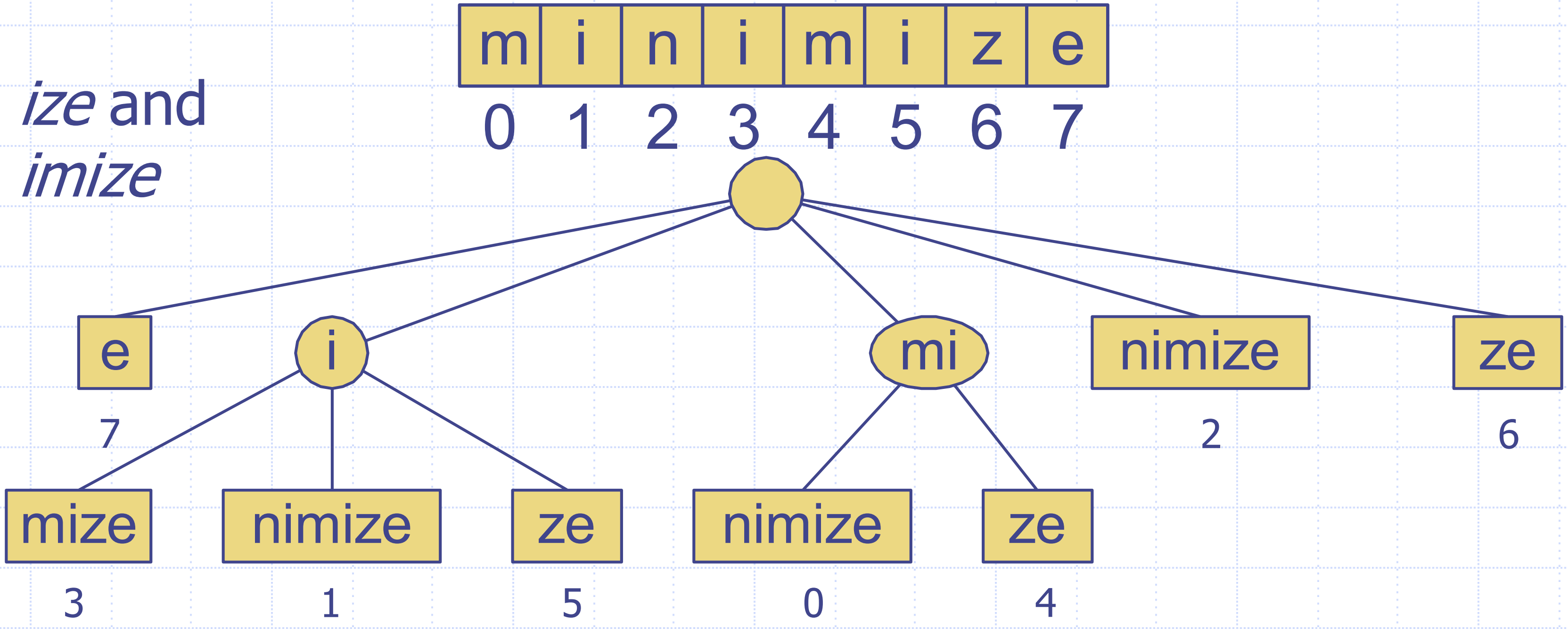
- Compact representation of the suffix trie for a string X of size n from an alphabet of size d
  - Uses $O(n)$ space
  - Supports arbitrary pattern matching queries in X in $O(dm)$ time, where m is the size of the pattern
  - Can be constructed in $O(n)$ time



Prefix Matching using Suffix Trie

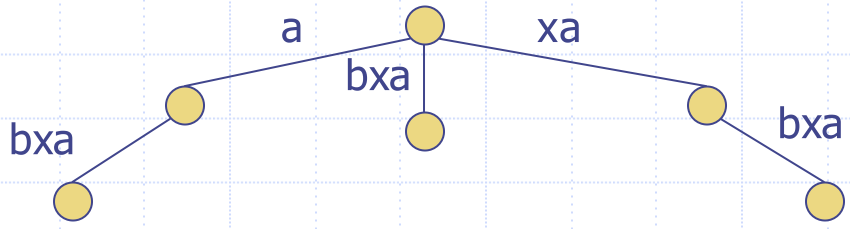
- If two suffixes have a same prefix, then their corresponding paths are the same at their beginning, and the concatenation of the edge labels of the mutual part is the prefix.

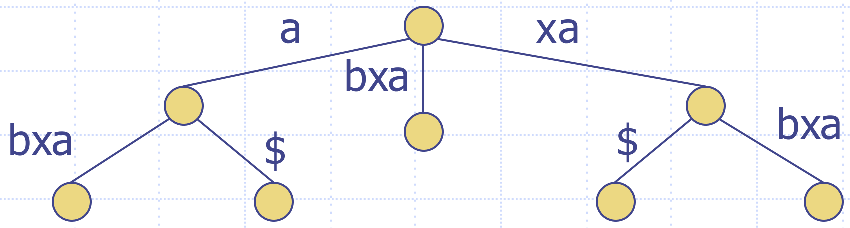
ize and
imize



Suffix Trie

- Not all strings are guaranteed to have corresponding suffix trie.
- For example: xabxa

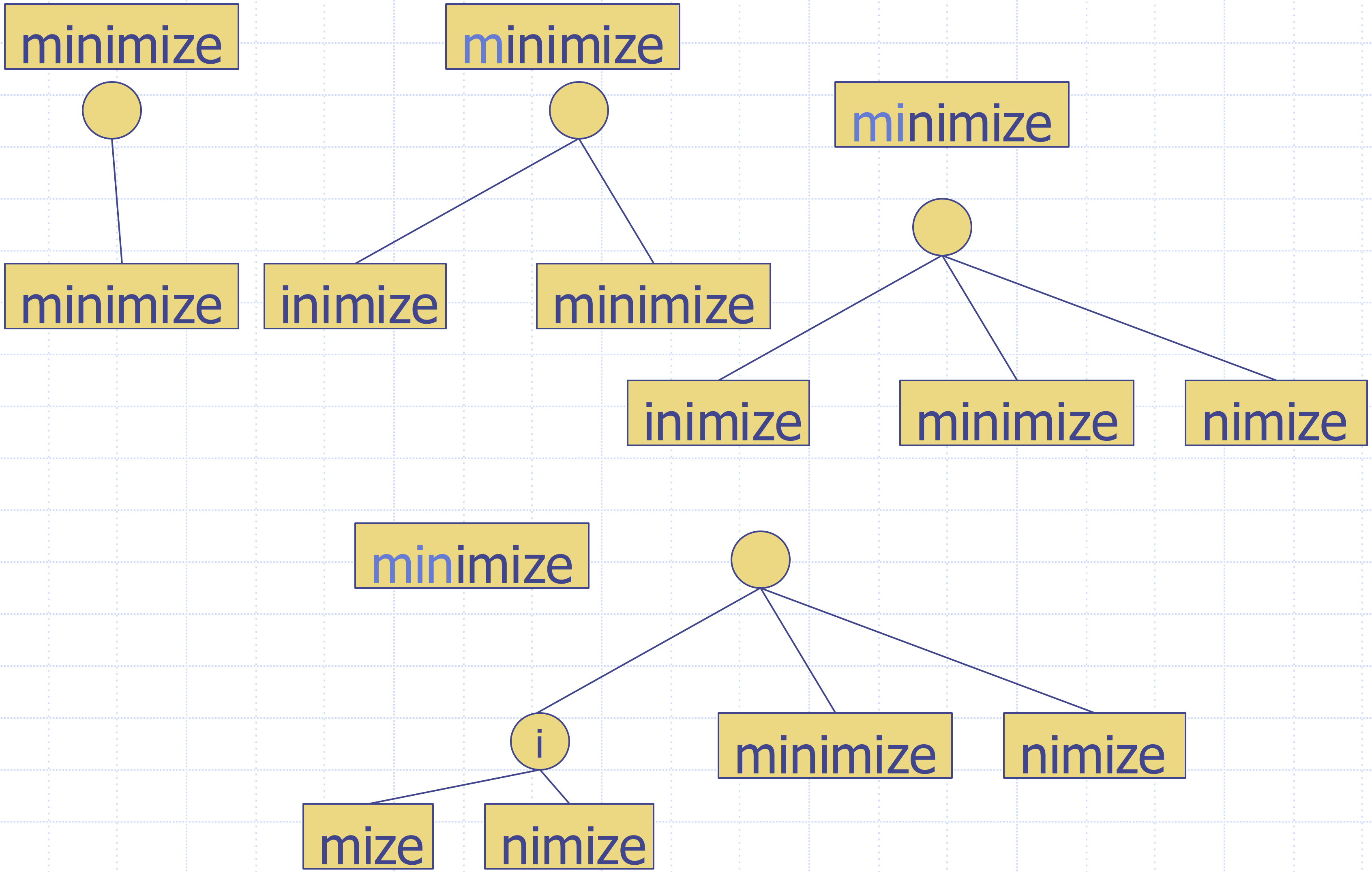




Constructing a Suffix Trie

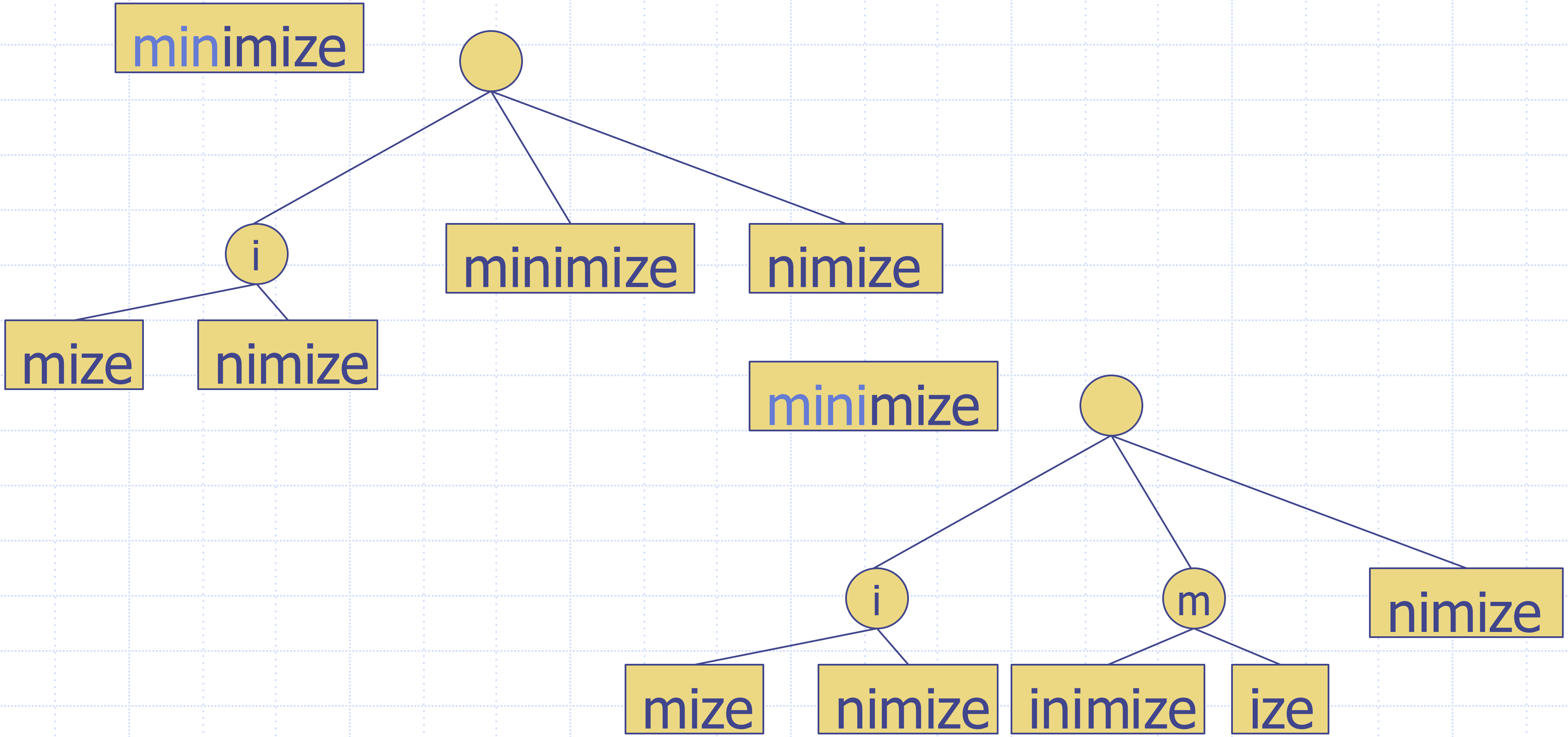
- S[1..n] is the string
- start with a single edge for S
- enter the edges for the suffix S[i..n] where i goes from 2 to n
  - Starting at the root node find the longest part from the root whose label matches a prefix of S[i..n]. At some point, no further matches are possible
    - If the point is at a node, then denote this node by w
    - If it is in the middle of an edge, then insert a new node called w, at this point
    - create a new edge running from w to a new leaf labeled S[i..n]

Example

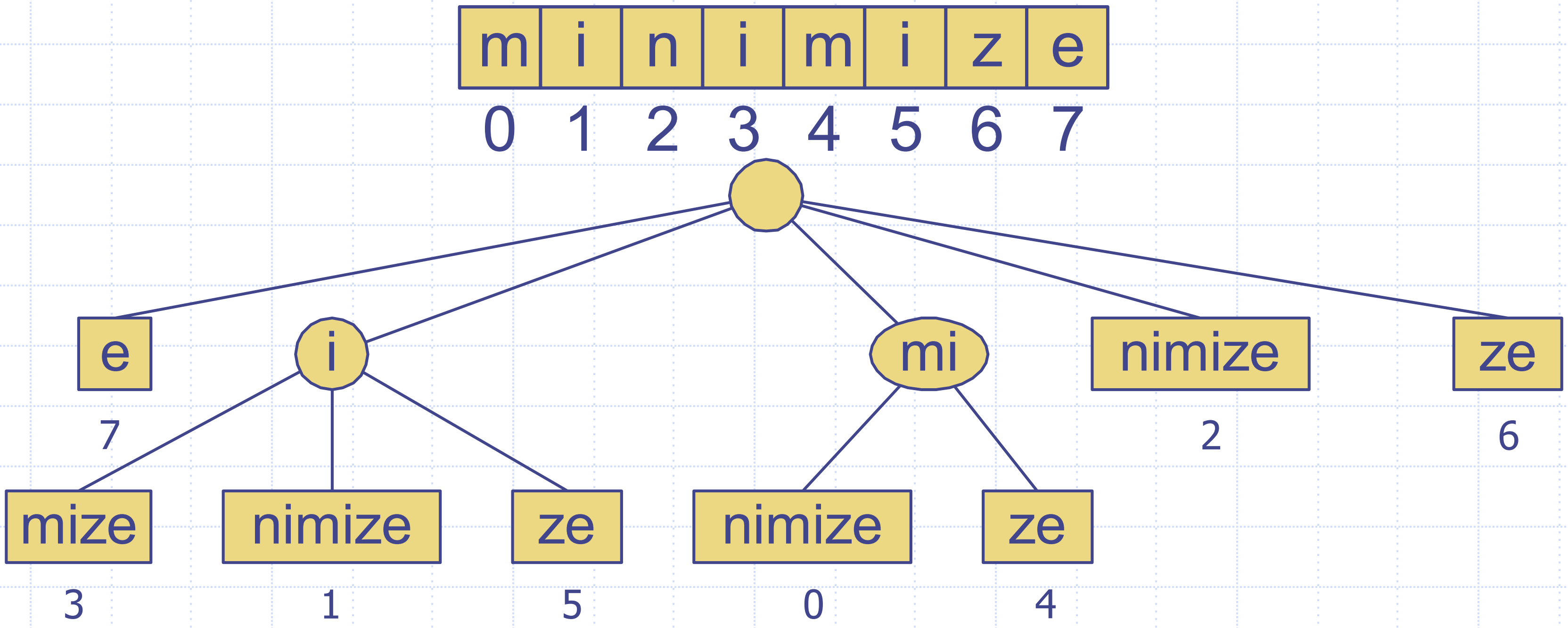


minimize
minimize
minimize
inimize
minimize
minimize
inimize
minimize
nimize
minimize
i
minimize
nimize
mize
nimize

Example (2)


minimize
i
minimize
nimize
mize
nimize
minimize
i
m
nimize
mize
nimize
inimize
ize

Example (3)


m i n i m i z e
0 1 2 3 4 5 6 7
e
7
i
mi
nimize
2
ze
6
mize
3
nimize
1
ze
5
nimize
0
ze
4

Constructing a Suffix Trie

- S[1..n] is the string
- start with a single edge for S
- enter the edges for the suffix S[i..n] where i goes from 2 to n
  - Starting at the root node find the longest part from the root whose label matches a prefix of S[i..n]. At some point, no further matches are possible
    - If the point is at a node, then denote this node by w
    - If it is in the middle of an edge, then insert a new node called w, at this point
    - create a new edge running from w to a new leaf labeled S[i..n]

Complexity- $O(n^2)$

Dynamic Programming

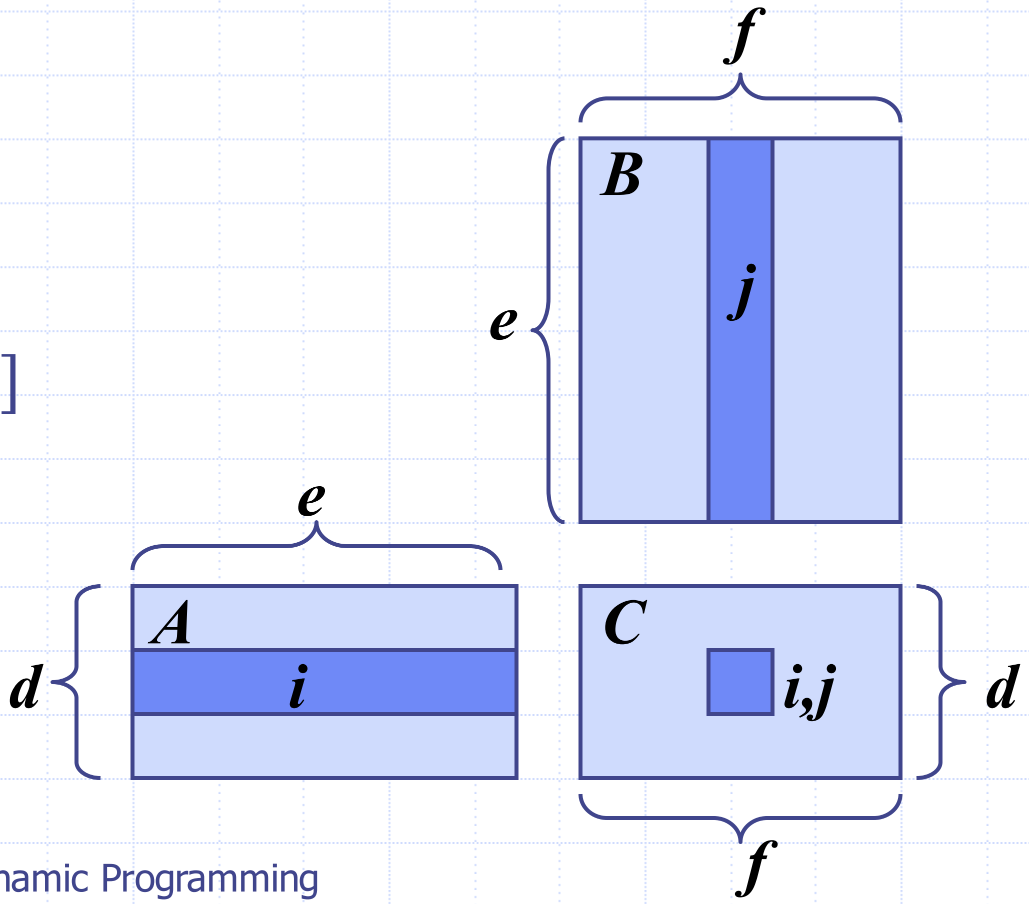
Motivating Example

- Fibonacci number computation
  - Recursive definition
    - fib(n) = fib(n-1)+fib(n-2)
  - Complexity – $O(\Phi^n)$

Matrix Chain-Products

- Dynamic Programming is a general algorithm design paradigm.
  - **Matrix Chain-Products**
- Review: Matrix Multiplication.
  - $C = A*B$
  - A is $d \times e$ and B is $e \times f$
  - $O(def)$ time

$$C[i,j] = \sum_{k=0}^{e} A[i,k]B[k,j]$$



Matrix Chain-Products

- **Matrix Chain-Product:**
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- Example
  - B is 3×100
  - C is 100×5
  - D is 5×5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

An Enumeration Approach

- **Matrix Chain-Product Alg.:**
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best
- Running time:
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

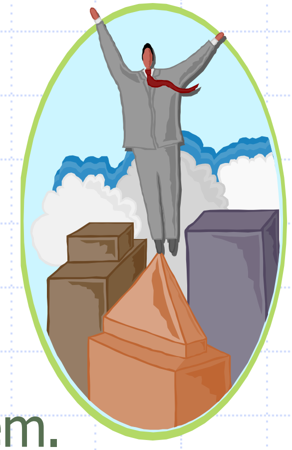
A Greedy Approach

- Idea #1: repeatedly select the product that uses (up) the most operations.
- Counter-example:
  - A is 10×5
  - B is 5×10
  - C is 10×5
  - D is 5×10
  - Greedy idea #1 gives (A*B)*(C*D), which takes 500+1000+500 = 2000 ops
  - A*((B*C)*D) takes 500+250+250 = 1000 ops

Another Greedy Approach

- Idea #2: repeatedly select the product that uses the fewest operations.
- Counter-example:
  - A is 101×11
  - B is 11×9
  - C is 9×100
  - D is 100×99
  - Greedy idea #2 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
- The greedy approach is not giving us the optimal value.

A "Recursive" Approach



- Define **subproblems**:
  - Find the best parenthesization of $A_i*A_{i+1}*\ldots*A_j$.
  - Let $N_{i,j}$ denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is $N_{0,n-1}$.
- **Subproblem optimality**: The optimal solution can be defined in terms of optimal subproblems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say, the final multiply is at index i: $(A_0*\ldots*A_i)*(A_{i+1}*\ldots*A_{n-1})$.
  - Then the optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i}$ and $N_{i+1,n-1}$ plus the time for the last multiply.
  - If the global optimum did not have these optimal subproblems, we could define an even better "optimal" solution.

A Characterizing Equation

- The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiply is at.
- Let us consider all possible places for that final multiply:
  - Recall that A_i is a $d_i \times d_{i+1}$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

- Note that subproblems are not independent--the **subproblems overlap**.

A Dynamic Programming Algorithm

- Since subproblems overlap, we don't use recursion.
- Instead, we construct optimal subproblems "bottom-up."
- $N_{i,i}$'s are easy, so start with them
- Then do length 2,3,… subproblems, and so on.
- The running time is $O(n^3)$

Algorithm ***matrixChain***(***S***):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal paranethization of S

for $i \leftarrow 1$ **to** $n-1$ **do**

$\quad N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ **to** $n-1$ **do**

$\quad$ **for** $i \leftarrow 0$ **to** $n-b-1$ **do**

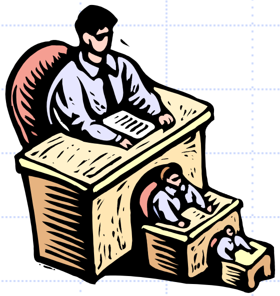
$\quad\quad j \leftarrow i+b$

$\quad\quad N_{i,j} \leftarrow +$**infinity**

$\quad\quad$ **for** $k \leftarrow i$ **to** $j-1$ **do**

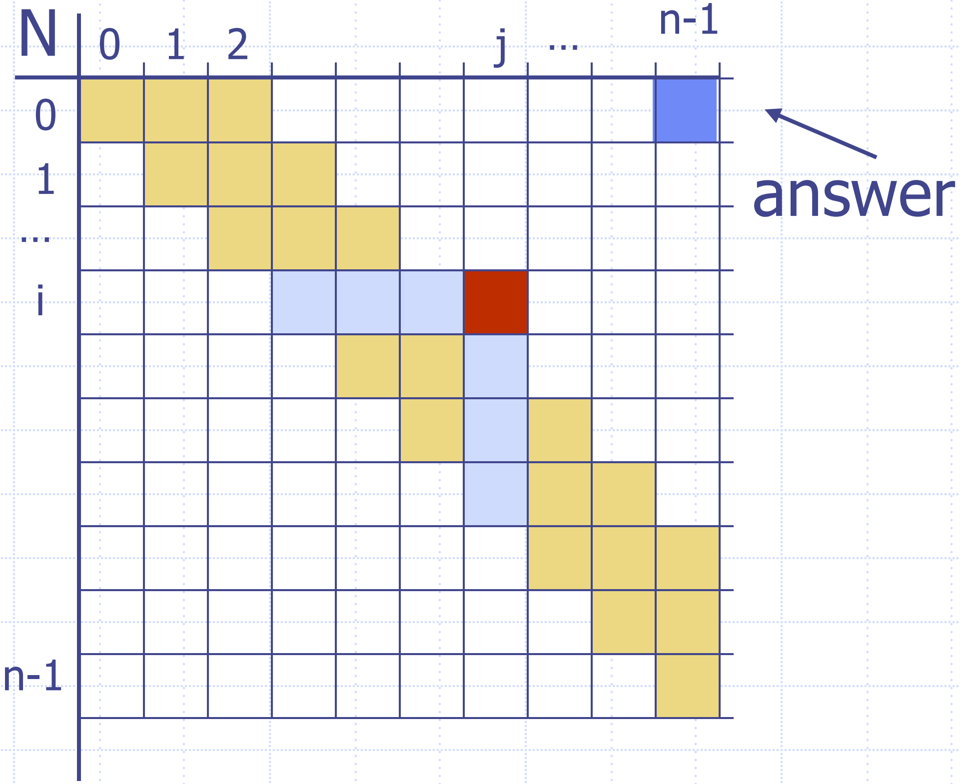
$\quad\quad\quad N_{i,j} \leftarrow \mathbf{min}\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

A Dynamic Programming Algorithm Visualization

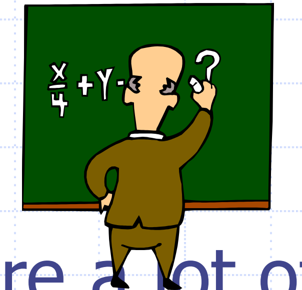


- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from pervious entries in i-th row and j-th column
- Filling in each entry in the N table takes O(n) time.
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering "k" for each N entry

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$



The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

Subsequences

- A ***subsequence*** of a character string $x_0x_1x_2\ldots x_{n-1}$ is a string of the form $x_{i_1}x_{i_2}\ldots x_{i_k}$, where $i_j < i_{j+1}$.
- Not the same as substring!
- Example String: ABCDEFGHIJK
  - Subsequence: ACEGIJK
  - Subsequence: DFGHK
  - Not subsequence: DAGH

The Longest Common Subsequence (LCS) Problem

- Given two strings X and Y, the longest common subsequence (LCS) problem is to find a longest subsequence common to both X and Y
- Has applications to DNA similarity testing (alphabet is {A,C,G,T})
- Example: ABCDEFG and XZACKDFWGH have ACDFG as a longest common subsequence

A Poor Approach to the LCS Problem

- A Brute-force solution:
  - Enumerate all subsequences of X
  - Test which ones are also subsequences of Y
  - Pick the longest one.
- Analysis:
  - If X is of length n, then it has 2^n subsequences
  - This is an exponential-time algorithm!