COMP9024: Data Structures and Algorithms

Week Ten: Text Processing

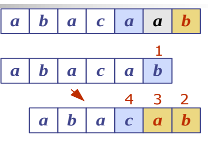
Hui Wu

Session 1, 2015
http://www.cse.unsw.edu.au/~cs9024

Outline

- Pattern Matching
- Tries
- The Greedy Method and Text Compression
- Dynamic Programming

Pattern Matching



Strings

- A string is a sequence of characters
- Examples of strings:
  - Java program
  - HTML document
  - DNA sequence
  - Digitized image
- An alphabet Σ is the set of possible characters for a family of strings
- Example of alphabets:
  - ASCII
  - Unicode
  - {0, 1}
  - {A, C, G, T}
- Let P be a string of size m
  - A substring $P[i .. j]$ of P is the subsequence of P consisting of the characters with ranks between i and j
  - A prefix of P is a substring of the type $P[0 .. i]$
  - A suffix of P is a substring of the type $P[i .. m - 1]$
- Given strings T (text) and P (pattern), the pattern matching problem consists of finding a substring of T equal to P
- Applications:
  - Text editors
  - Search engines
  - Biological research

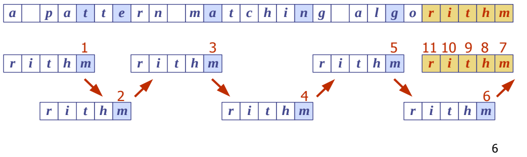
Brute-Force Pattern Matching

- The brute-force pattern matching algorithm compares the pattern P with the text T for each possible shift of P relative to T, until either
  - a match is found, or
  - all placements of the pattern have been tried
- Brute-force pattern matching runs in time $O(nm)$
- Example of worst case:
  - $T = aaa \ldots ah$
  - $P = aaah$
  - may occur in images and DNA sequences
  - unlikely in English text

```
Algorithm BruteForceMatch(T, P)
  Input text T of size n and pattern
      P of size m
  Output starting index of a substring of
      T equal to P or −1 if no such
      substring exists
{ for ( i = 0; i < n – m+1; i++ )
   { // test shift i of the pattern
     j = 0;
     while ( j < m ∧ T[i + j] = P[j] )
       j = j + 1;
     if ( j = m )
       return i; // match at i
   }
  return -1 // no match anywhere
}
```


Boyer-Moore Heuristics

- The Boyer-Moore's pattern matching algorithm is based on two heuristics
  Looking-glass heuristic: Compare P with a subsequence of T moving backwards
  Character-jump heuristic: When a mismatch occurs at $T[i] = c$
  - If P contains c, shift P to align the last occurrence of c in P with $T[i]$
  - Else, shift P to align $P[0]$ with $T[i + 1]$
- Example



Last-Occurrence Function

- Boyer-Moore's algorithm preprocesses the pattern P and the alphabet Σ to build the last-occurrence function L mapping Σ to integers, where $L(c)$ is defined as
  - the largest index i such that $P[i] = c$ or
  - -1 if no such index exists
- Example:
  - $\Sigma = \{a, b, c, d\}$
  - $P = abacab$

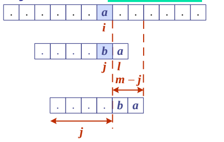
c	a	b	c	d
$L(c)$	4	5	3	-1

- The last-occurrence function can be represented by an array indexed by the numeric codes of the characters
- The last-occurrence function can be computed in time $O(m + s)$, where m is the size of P and s is the size of Σ

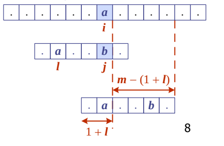
The Boyer-Moore Algorithm

```
Algorithm BoyerMooreMatch(T, P, Σ)
  { L = lastOccurenceFunction(P, Σ )
    i = m – 1
    j = m – 1
    repeat
      if ( T[i] = P[j] )
        { if ( j = 0 )
            return i  // match at i
          else
            { i = i – 1;
              j = j – 1; }
        }
      else  // character-jump
        { l = L[T[i]];
          i = i + m – min(j, 1 + l);
          j = m – 1; }
    until ( i > n – 1)
    return –1  //  no match
  }
```

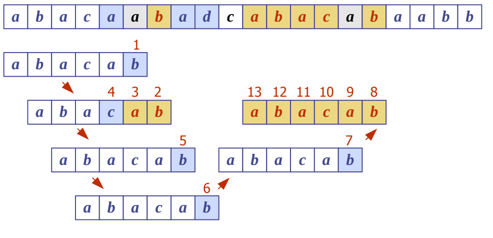
Case 1: $j \leq 1 + l$



Case 2: $1 + l \leq j$

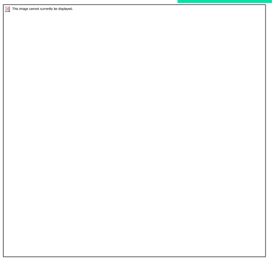


Example



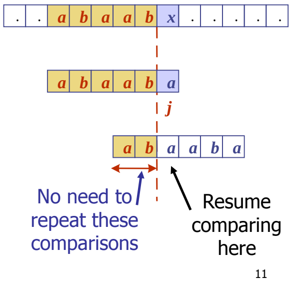
Analysis

- Boyer-Moore's algorithm runs in time $O(nm + s)$
- Example of worst case:
  - $T = aaa \ldots a$
  - $P = baaa$
- The worst case may occur in images and DNA sequences but is unlikely in English text
- Boyer-Moore's algorithm is significantly faster than the brute-force algorithm on English text



The KMP Algorithm

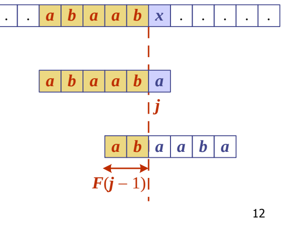
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$



KMP Failure Function

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself
- The **failure function** $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j - 1)$

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3

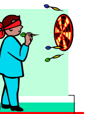


The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j-1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP's algorithm runs in optimal time $O(m + n)$

```
Algorithm KMPMatch(T, P)
  { F = failureFunction(P);
    i = 0;
    j = 0;
    while ( i < n )
      if ( T[i] = P[j] )
        { if ( j = m – 1 )
              return  i – j ; // match
          else
              { i = i + 1; j = j + 1;}
        }
      else
        if ( j > 0 )
            j = F[j – 1];
        else
            i = i + 1;
      return  –1;  // no match
  }
```

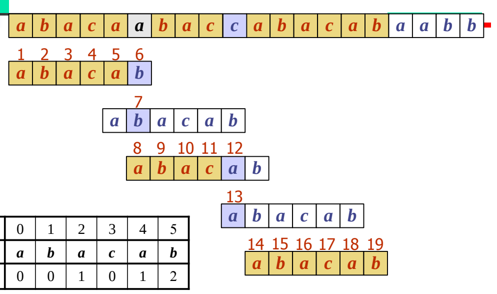

Computing the Failure Function



- The failure function can be represented by an array and can be computed in $O(m)$ time
- The construction is similar to the KMP algorithm itself
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j-1) < j$)
- Hence, there are no more than $2m$ iterations of the while-loop

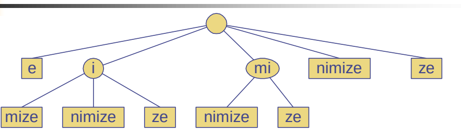
```
Algorithm failureFunction(P)
{  F[0] = 0;
   i = 1;
   j = 0;
   while  ( i < m )
     if  ( P[i] = P[j] )
       { // we have matched j + 1 char
         F[i] =  j + 1;
         i = i + 1;
         j = j + 1; }
     else if  (  j > 0 )
         // use failure function to shift P
           j = F[j – 1];
         else
           { F[i] = 0; //  no match
             i = i + 1;}
}
```


Example



j	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	b
$F(j)$	0	0	1	0	1	2

Tries



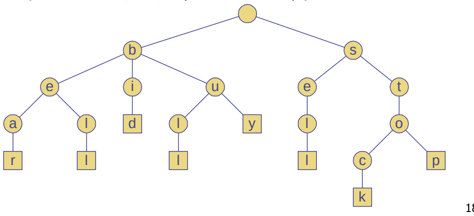
Preprocessing Strings

- Preprocessing the pattern speeds up pattern matching queries
  - After preprocessing the pattern, KMP's algorithm performs pattern matching in time proportional to the text size
- 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 tries 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

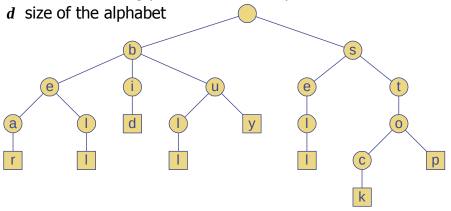
  S = { bear, bell, bid, bull, buy, sell, stock, stop }



Analysis of Standard Tries

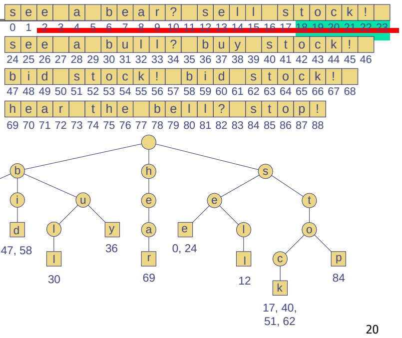
A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:

- n total size of the strings in S
- m size of the string parameter of the operation
- d size of the alphabet



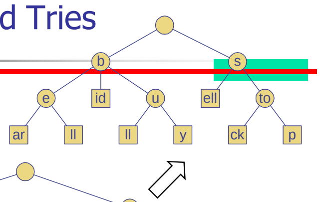
Word Matching with a Trie

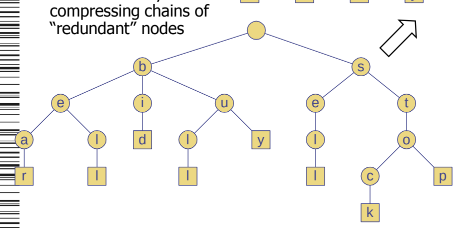
- We insert the words of the text into a trie
- Each leaf stores the occurrences of the associated word in the text



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

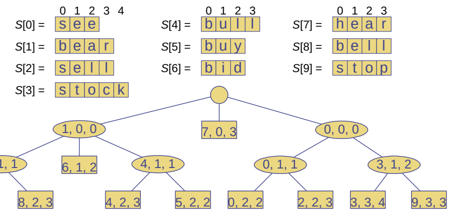




Compact Representation

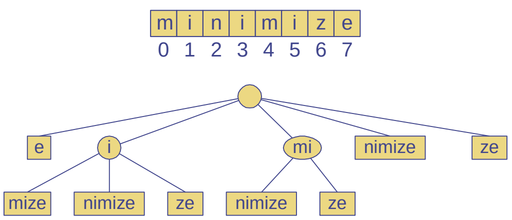
Compact representation of a compressed trie for an array of strings:

- 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



Suffix Trie

- The suffix trie of a string X is the compressed trie of all the suffixes of X



Pattern Matching Using Suffix Trie (1/2)

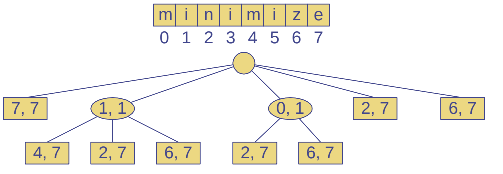
```
Algorithm suffixTrieMatch(T, P)
{   p = P.length;   j = 0; v = T.root();
   repeat
     {  f = true;
        for  each child  w of v  do
          { // we have matched j + 1 char
             i = start(w);    // start(w) is the start index of w
             if ( P[j] = T[i] )    // process child w
               {  x = end(w) – i +1;   // end(w) is the end index of w
                  if  (  p ≤ x )
                   // suffix is shorter than or of the same length of the node label
                    { if  (  P[j:j+p–1] = X[i:i+p–1]  )   return i – j ;
                      else return "P is not a substring of X" ; }
                  else // suffix is longer than the node label
                    { if  (  P[j:j+x–1] = X[i:i+x–1]  )
                         {  p = p – x;  // update suffix length
                            j = j + x;  // update suffix start index
                            v = w;  f = false;  break out of the for loop; } } } } }
   until  f or T.isExternal(v);
   return "P is not a substring of X" ;
 }
```

Pattern Matching Using Suffix Trie (2/2)

- Input of the algorithm:
  - Compact suffix trie T for a text X and pattern P.
- Output of the algorithm:
  - Starting index of a substring of X matching P or an indication that P is not a substring.
- The algorithm assumes the following additional property on the labels of the nodes in the compact representation of the suffix trie:
  - If node v has label (i, j) and Y is the string of length y associated with the path from the root to v (included), then X[j-y+1..j]=Y.
- This property ensures that we can easily compute the start index of the pattern in the text when a match occurs.

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



The Greedy Method and Text Compression

The Greedy Method Technique

- **The greedy method** is a general algorithm design paradigm, built on the following elements:
  - **configurations**: different choices, collections, or values to find
  - **objective function**: a score assigned to configurations, which we want to either maximize or minimize
- It works best when applied to problems with the **greedy-choice** property:
  - a globally-optimal solution can always be found by a series of local improvements from a starting configuration.

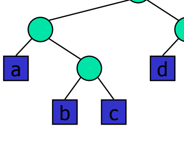
Text Compression

- Given a string X, efficiently encode X into a smaller string Y
  - Saves memory and/or bandwidth
- A good approach: **Huffman encoding**
  - Compute frequency f(c) for each character c.
  - Encode high-frequency characters with short code words
  - No code word is a prefix for another code
  - Use an optimal encoding tree to determine the code words

Encoding Tree Example

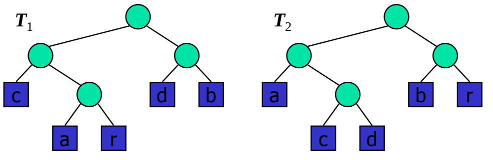
- A **code** is a mapping of each character of an alphabet to a binary code-word
- A **prefix code** is a binary code such that no code-word is the prefix of another code-word
- An **encoding tree** represents a prefix code
  - Each external node stores a character
  - The code word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

00	010	011	10	11
a	b	c	d	e



Encoding Tree Optimization

- Given a text string X, we want to find a prefix code for the characters of X that yields a small encoding for X
  - Frequent characters should have short code-words
  - Rare characters should have long code-words
- Example
  - X = abracadabra
  - T_1 encodes X into 29 bits
  - T_2 encodes X into 24 bits



Huffman's Algorithm

- Given a string X, Huffman's algorithm construct a prefix code the minimizes the size of the encoding of X
- It runs in time $O(n + d \log d)$, where n is the size of X and d is the number of distinct characters of X
- A heap-based priority queue is used as an auxiliary structure

Algorithm *HuffmanEncoding*(X)
  Input string X of size n
  Output optimal encoding trie for X

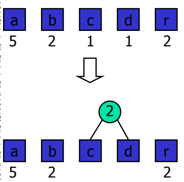
```
{
  C = distinctCharacters(X);
  computeFrequencies(C, X);
  Q = new empty heap;
  for all c ∈ C
    { T = new single-node tree storing c;
      Q.insert(getFrequency(c), T); }
  while ( Q.size() > 1 )
    {  f1 = Q.minKey();
       T1 = Q.removeMin();
       f2 = Q.minKey();
       T2 = Q.removeMin();
       T = join(T1, T2);
       Q.insert(f1 + f2, T);
    }
  return Q.removeMin();
}
```

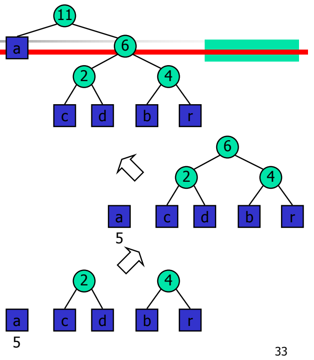

Example

X = abracadabra

Frequencies

a	b	c	d	r
5	2	1	1	2

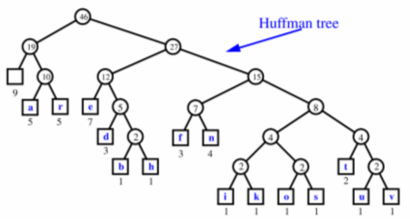




Extended Huffman Tree Example

String: a fast runner need never be afraid of the dark

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

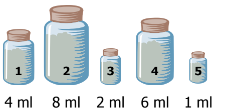


The Fractional Knapsack Problem (not in book)

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.
  - In this case, we let x_i denote the amount we take of item i
  - Objective: maximize $\sum_{i \in S} b_i (x_i / w_i)$
  - Constraint: $\sum_{i \in S} x_i \le W$

Example

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.



Items:	1	2	3	4	5
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	\$12	\$32	\$40	\$30	\$50
Value: (\$ per ml)	3	4	20	5	50

"knapsack"

10 ml

Solution:
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

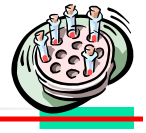
The Fractional Knapsack Algorithm

- Greedy choice: Keep taking item with highest **value** (benefit to weight ratio)
  - Since $\sum_{i \in S} b_i (x_i / w_i) = \sum_{i \in S} (b_i / w_i) x_i$
  - Run time: O(n log n). Why?
- Correctness: Suppose there is a better solution
  - there is an item i with higher value than a chosen item j, but $x_i < w_i$, $x_j > 0$ and $v_i < v_j$
  - If we substitute some i with j, we get a better solution
  - How much of i: $\min\{w_i - x_i, x_j\}$
  - Thus, there is no better solution than the greedy one

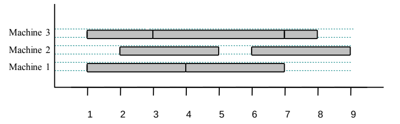
Algorithm *fractionalKnapsack(S, W)*
Input: set S of items with benefit b_i and weight w_i; max. weight W
Output: amount x_i of each item i to maximize benefit with weight at most W

```
{ for each item i in S
    { x_i = 0;
      v_i = b_i / w_i ;   // value
    }
  w = 0;                // total weight
  while ( w < W )
    { remove item i with highest v_i
      x_i = min{w_i , W - w};
      w = w + min{w_i , W - w};
    }
}
```


Task Scheduling (not in book)



- Given: a set T of n tasks, each having:
  - A start time, s_i
  - A finish time, f_i (where $s_i < f_i$)
- Goal: Perform all the tasks using a minimum number of "machines."



Task Scheduling Algorithm

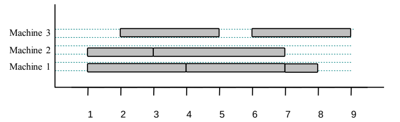
- Greedy choice: consider tasks by their start time and use as few machines as possible with this order.
  - Run time: O(n log n). Why?
- Correctness: Suppose there is a better schedule.
  - We can use k-1 machines
  - The algorithm uses k
  - Let i be first task scheduled on machine k
  - Machine i must conflict with k-1 other tasks
  - But that means there is no non-conflicting schedule using k-1 machines

Algorithm *taskSchedule(T)*
Input: set T of tasks with start time s_i and finish time f_i
Output: non-conflicting schedule with minimum number of machines

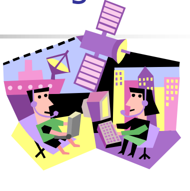
```
{ m = 0;   // no. of machines
  while T is not empty
    { remove task i with smallest s_i
      if there's a machine j for i then
        schedule i on machine j ;
      else
        { m = m + 1;
          schedule i on machine m;
        }
    }
}
```


Example

- Given: a set T of n tasks, each having:
  - A start time, s_i
  - A finish time, f_i (where $s_i < f_i$)
  - [1,4], [1,3], [2,5], [3,7], [4,7], [6,9], [7,8] (ordered by start)
- Goal: Perform all tasks on min. number of machines



Dynamic Programming

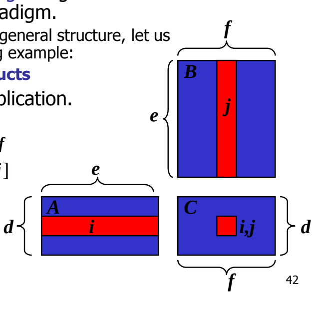


Matrix Chain-Products (not in book)

- Dynamic Programming is a general algorithm design paradigm.
  - Rather than give the general structure, let us first give a motivating example:
  - **Matrix Chain-Products**
- Review: Matrix Multiplication.
  - $C = A * B$
  - A is $d \times e$ and B is $e \times f$

$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$

  - $O(def)$ time



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

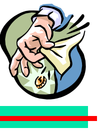
43

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:
  - The number of parenthesizations is equal to the number of binary trees with n nodes
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

44

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

45

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.

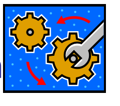
46

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.

47

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:

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

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

48

A Dynamic Programming Algorithm

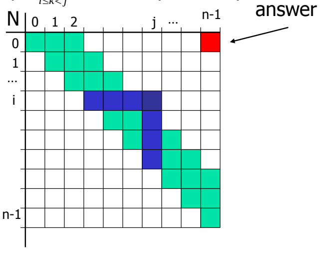
- Since subproblems overlap, we don't use recursion.
- Instead, we construct optimal subproblems "bottom-up."
- $N_{i,i}$'s are easy, so start with them
- Then do length 2,3,… subproblems, and so on.
- Running time: $O(n^3)$

```
Algorithm matrixChain(S):
  Input: sequence S of n matrices to be multiplied
  Output: number of operations in an optimal
     parenthesization of S
 { for ( i = 1; i ≤ n-1; i++ )
      N_{i,i} = 0;
    for ( b = 1; b ≤ n-1; b++ )
      for ( i = 0; i ≤ n-b-1; i++ )
        { j = i+b;
          N_{i,j} = +infinity;
          for ( k = i ; k ≤ j-1; i++ )
            N_{i,j} = min{N_{i,j} , N_{i,k} +N_{k+1,j} +d_i d_{k+1}
  d_{j+1}};
        }
 }
```

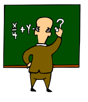

A Dynamic Programming Algorithm Visualization

- 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 running 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: ACEGJIK
  - 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

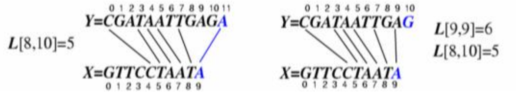
A Poor Approach to the LCS Problem

- 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!

A Dynamic-Programming Approach to the LCS Problem

- Define L[i,j] to be the length of the longest common subsequence of X[0..i] and Y[0..j].
- Allow for -1 as an index, so L[-1,k] = 0 and L[k,-1]=0, to indicate that the null part of X or Y has no match with the other.
- Then we can define L[i,j] in the general case as follows:
    1. If $x_i = y_j$, then L[i,j] = L[i-1,j-1] + 1 (we can add this match)
    2. If $x_i \neq y_j$, then L[i,j] = max{L[i-1,j], L[i,j-1]} (we have no match here)

Case 1: Case 2:



An LCS Algorithm

```
Algorithm LCS(X,Y):
  Input: Strings X and Y with n and m elements, respectively
  Output: For i = 0,…,n-1, j = 0,...,m-1, the length L[i, j] of a longest
    string that is a subsequence of both the string X[0..i] = x0x1x2…xi and
    the string Y[0.. j] = y0y1y2…yj
 { for ( i =1; i ≤ n-1, i ++ )
      L[i,-1] = 0;
   for ( j =0; i ≤ m-1, j ++ )
      L[-1,j] = 0;
   for ( i =0; i ≤ n-1, i ++ )
      for ( j =0; i ≤ m-1, j ++ )
         if ( xi = yj )
            L[i, j] = L[i-1, j-1] + 1;
         else
            L[i, j] = max{L[i-1, j] , L[i, j-1]};
   return array L;
 }
```


Visualizing the LCS Algorithm

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	1	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	2	2	2	3	4	4	4	4	4
6	0	1	1	2	2	3	3	3	4	4	5	5	5
7	0	1	1	2	2	3	4	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	5	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	6

Analysis of LCS Algorithm

- We have two nested loops
    - The outer one iterates *n* times
    - The inner one iterates *m* times
    - A constant amount of work is done inside each iteration of the inner loop
    - Thus, the total running time is O(*nm*)
- Answer is contained in L[n,m] (and the subsequence can be recovered from the L table).

References

1. Chapter 12, Data Structures and Algorithms by Goodrich and Tamassia.