More Stuff

Traveling Salesman Problem

- Input: Undirected weighted graph G = (V,E). Let W(e) denote the cost of edge e.
- Output: A tour P with minimum total cost. (A tour is a cycle P that visits all vertices exactly once). That is: for all edges e in tour P, minimize

$$W(P) = \sum_{e \in P} W(e)$$

General Backtrack Search Skeleton

- BacktrackOptimalSearch(// very rough outline

```
Let move1, move2, … movek represent the k
   possible ways of making the next step.
For each possible way movei
      try movei.
      assuming movei was done,
         make recursive call to find best solution
         on smaller subproblem
      overall solution cost =
         best subproblem solution + cost (movei)
      keep track of best overall cost so far
return best overall cost found.
```

line-breaking problem

- Given sequence of words from one paragraph
- Return where line-breaks should occur
- Minimize empty space on each line (except for last line of paragraph)

line-breaking problem

- A simple version:
  - letters and spaces have equal width
  - input is set of n word lengths, w_1, w_2, ... w_n
  - also given line width limit L.
  - each length w_i includes one space
  - Placing words i up to j on one line means
    $$\sum_{k=i}^{j} w_i \leq L$$
  - Penalty for extra spaces $X = L - \sum_{k=i}^{j} w_i$ is X^3
  - Minimize sum of penalties from each line (no last line penalty)

Recursive Backtrack Search

- Let w[] be array of lengths of n words; L is line width
- Compute lineBreak(0) to solve linebreaking problem.
- Algorithm lineBreak(i) {

```
Input: Integer i indicating which word subproblem starts at.
Output: returns minimum total penalty when
        placing w[i], w[i+1], … w[n-1] into lines
  if (w[i] + w[i+1] + … + w[n-1] < L) return 0;
  mincost ← Infinity;
  k ← 1;
  while (k words starting from w[i] fit on a line)
          // meaning (w[i] + w[i+1] + … + w[i+k-1] <= L)
    linecost ← penalty from placing words w[i] to w[i+k-1]
                on one line.
    totalcost ← linecost + lineBreak(i+k);
    mincost ← min(totalcost, mincost)  // track minimum so far
    k++;
  return mincost;
```

Example problem

- Paragraph is:
  Those who cannot remember the past are condemned to repeat it.
- Word lengths are 6,4,7,9,4,5,4,10,3,7,4.
- Suppose line width L = 17.
- Find an optimal way of separating words into lines that minimizes penalty.

Greedy method

- Input:
  - int [] w : array of word lengths.
  - int n : length of w.
  - int L : line length
- Output:
  - int [] LastWord : array for storing last word on each line.
    LastWord[i] is the index of the last word stored on line i.
  - // start counting arrays at index 0.

Dynamic Programming

- DP version of Recursive backtrack LineBreak problem
  - Use array lineB[] to store subproblem costs
  - lineB[i] is min cost of linebreaking solution for words (w[i], w[i+1], … w[n-1]).
  - compute lineB in reverse order (from n-1 down to 0).

linebreak DP

```
for i ← n-1 downto 0 do
  if (w[i] + w[i+1] + … + w[n-1] < L)
     lineB[i] ← 0;
  else
     mincost ← Infinity;
     k ← 1;
     while (k words starting from w[i] fit on a line)
               // meaning (w[i] + w[i+1] + … + w[i+k-1] <= L)
        linecost ← penalty from placing words w[i] to w[i+k-1]
                on one line.
        totalcost ← linecost + lineB[i+k];
        mincost ← min(totalcost, mincost)  // track min. so far
        k++;
     lineB[i]=mincost;
```

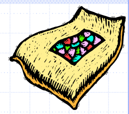

linebreak DP cost

- O(nL); L is maximum width
- Linear if L is considered constant
- Space O(n).

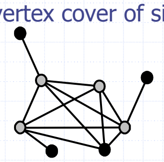
Longest Common Subsequence

- Given : two strings A & B
- Find longest common (possibly non-contiguous) subsequence
  - Here, subsequence ≠ substring
  - Example: A= "R8D4F7G"
    B= "4RD97G2"
    answer is "RD7G"

Vertex Cover



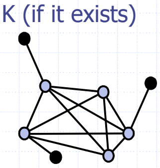
- A vertex cover of graph G=(V,E) is a subset W of V, such that, for every edge (a,b) in E, a is in W or b is in W.
- VERTEX-COVER: Given an graph G and an integer K, return a vertex cover of size K (if it exists)



Clique

- A **clique** of a graph G=(V,E) is a subgraph C that is fully-connected (every pair in C has an edge).
- CLIQUE: Given a graph G and an integer K, return a clique in G of size K (if it exists)

This graph has
a clique of size 5



Some Other Problems

- SET-COVER: Given a collection of m sets, and an integer K, pick K of the sets such that the union of the K sets is the same as the union of the whole collection of m sets.
- SUBSET-SUM: Given a set of integers and an integer K, find a subset of the integers that sums to exactly K.
- 0/1 Knapsack: Given a collection of items with weights and benefits, find a subset of weight at most W and benefit at least K.
- Hamiltonian-Cycle: Given an graph G, find a cycle in G that visits each vertex exactly once

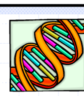
Outline and Reading

- Strings (§9.1.1)
- Pattern matching algorithms
  - Brute-force algorithm (§9.1.2)
  - Boyer-Moore algorithm (§9.1.3)
  - Knuth-Morris-Pratt algorithm (§9.1.4)

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 Algorithm



- 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

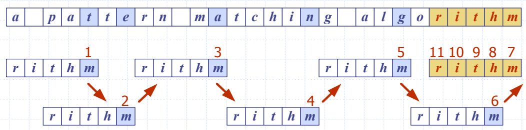
```
Algorithm BruteForceMatch(T, P)
  Input text T of size n and pattern
     P of size m
  Output starting index of a
     substring of T equal to P or −1
     if no such substring exists
  for i ← 0 to n − m
     { test shift i of the pattern }
     j ← 0
     while j < m ∧ T[i + j] = P[j]
        j ← j + 1
     if j = m
        return i {match at i}
     else
        break while loop {mismatch}
  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:
  - $\Sigma = \{a, b, c, d\}$
  - $P = abacab$

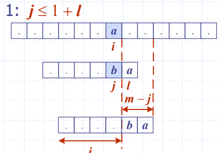
c	a	b	c	d
$L(c)$	4	5	3	-1

- 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

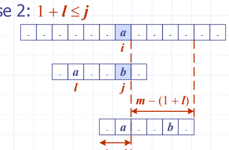
The Boyer-Moore Algorithm

```
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 }
```

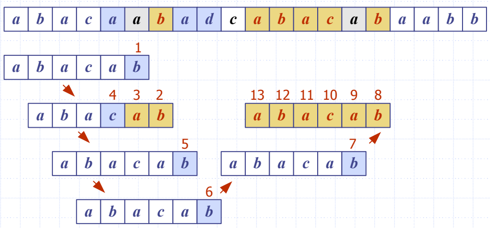
Case 1: $j \leq 1 + l$



Case 2: $1 + l \leq j$

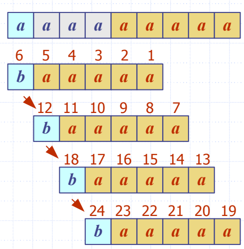


Example



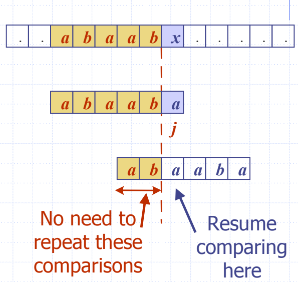
Analysis

- Boyer-Moore's algorithm runs in time $O(nm + s)$
- Example of worst case:
  - $T = aaa \dots a$
  - $P = baaa$
- 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 - Motivation

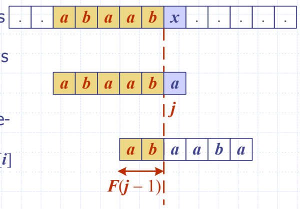
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix 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 $O(m+n)$

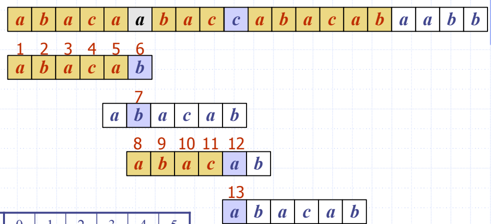
```
Algorithm KMPMatch(T, P)
  F ← failureFunction(P)
  i ← 0
  j ← 0
  while i < n
    if T[i] = P[j]
      if j = m − 1
        return i − j { match }
      else
        i ← i + 1
        j ← j + 1
    else
      if j > 0
        j ← F[j − 1]
      else
        i ← i + 1
  return −1 { no match }
```

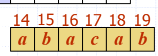

Computing the Failure Function

- 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 chars}
      F[i] ← j + 1
      i ← i + 1
      j ← j + 1
    else if j > 0 then
      {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