CSCI S-Q
Lecture #13
String Searching
8/3/98

• Administrivia
  • Final Exam - Wednesday 8/12, 6:15pm, SC102B
  • Room for class next Monday
  • Graduate Paper due Friday

• Tonight
  • Precomputation
  • Brute force string searching method
  • Knuth-Morris-Pratt algorithm
  • Boyer-Moore algorithm
  • Rabin-Karp algorithm
Precomputation

• One of the best ways to make a computation efficient is to avoid it altogether. In many situations, we can avoid performing a potentially lengthy computation through the use of extra memory, by storing information that would otherwise be costly to recompute.

• Most of the examples we’ve seen so far come from data structures:
  
  • Singly-Linked Lists
    Problem: Find predecessor (for deletion).
    Solution: Doubly-Linked lists. PREV is O(1).
  
  • List-Based Deques
    Problem: Keeping track of head and tail.
    Solution: Use a header structure.
  
• Tonight we’ll focus on algorithms and how they can benefit from these ideas.
Principles of Precomputation

1. Do whatever part of the computation you can ahead of time.
   
   Example:
   
   • Presorting the data (if it is available) or otherwise arranging it for fast access (as in a telephone book)

2. Invest/Plan ahead - try to avoid getting into situations that will cause problems later.
   
   Examples:
   
   • Don’t let binary search trees get too unbalanced - use balancing operations.
   • Rehash if load factor of "too high" or "too low" is detected.
   • Use amortized analysis to determine whether the investment is worthwhile.

3. Avoid recomputing things that you have already computed.

We will see several uses of precomputation tonight.
The Problem - String Search (P,T)

• Review

A string is a finite sequence of characters taken from a finite alphabet.

• The problem:

Given strings P (the Pattern) and T (the Text)

find the smallest $i$ (where $i$ is the shift) such that $T$ contains $P$ as a substring beginning at index $i$ (zero-based). If no such shift exists, return -1.

(Instead of finding the smallest, we could find the largest, $n^{th}$, all the matches, etc. We'll just focus on finding the smallest for now.)

Examples

STRINGSEARCH ("ABC", "XYZABCD") = 3

<table>
<thead>
<tr>
<th>pattern</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>XYZABCD</td>
</tr>
</tbody>
</table>

STRINGSEARCH ("XY", "XYZABCD") = 0
STRINGSEARCH ("XZ", "XYZABCD") = -1
Brute Force Algorithm

Let $M = |P|$ (the length of the pattern)
Let $N = |T|$ (the length of the text)

for ($i = 0; i <= N-M; i++$) {
  try matching $P$ with the substring of length $M$ starting at index $i$ in $T$.
  If successful, return $i$.
}

return (-1);

Note: it is very important that the substring comparison stop as soon as a mismatch is detected... otherwise this is very slow.
Analysis of Brute-Force Method

• For no match:
  • In the worst case the last character in the pattern will fail to match each substring of T, and there is no match. For example:

    \[ T = \text{AAAAAAA}....A \]
    \[ P = \text{AAAAB} \]

    The number of comparisons will be \((N-M+1)M\)

  • In the best case the first character in the pattern always mismatches, and there is no match. For example:

    \[ T = \text{AAAAAAA}....A \]
    \[ P = \text{BAAA} \]

    The number of comparisons will be \((N-M+1)\)

• Generally, the brute-force algorithm is \(O(NM)\), but as we’ve observed, the actual complexity depends a lot upon the pattern and the text. Finding the word "themes" is going to take a lot more time than "zebras" if \(T\) is the New York Times - lots of mismatches after "the", for example, but not many mismatches after "z".

• Our first task is to free ourselves from the tyranny of "good" and "bad" patterns. Our goal will be to only look at each character in the text once.
The Knuth-Morris-Pratt Algorithm:

An Observation

• When a mismatch is detected after several of the characters in the text have matched the pattern, then we know what those characters are (by virtue of their having matched)...

\[ i = 0 \quad T = \begin{array}{cccccc}
A & B & C & X & B & C \\
\end{array} \]

\[ P = \begin{array}{cccc}
A & B & C & D \\
\end{array} \]

mismatch

• The brute-force algorithm would just increment \( i \) and try again. We know better - because we matched three characters before the mismatch, we know that the text at this shift starts with "A B C". The only match for A is at the first shift, so we must shift past.
Observation (continued)

- Comparing against the known part of T, and ignoring for the moment that we have seen the character in T that caused the mismatch...

  \[
  \begin{align*}
  i = 1 & \quad T = \overbrace{A B C \ldots}^{A B C D} \quad \text{No...} \\
  & \quad P = \underbrace{A B C D}_{A B C D} \\
  i = 2 & \quad T = \overbrace{A B C \ldots}^{A B C D} \quad \text{No...} \\
  & \quad P = \underbrace{A B C D}_{A B C D} \\
  i = 3 & \quad T = \overbrace{A B C ? \ldots}^{A B C D} \quad \text{Maybe...} \\
  & \quad P = \underbrace{A B C D}_{A B C D}
  \end{align*}
  \]

- So we can increment i immediately by 3 positions.
More Observations

• Things are not always so easy...

\[
i = 0 \quad T = \begin{array}{c} \_A \_A \_A \_X \end{array} \\
P = \begin{array}{c} \_A \_A \_A \_B \end{array}
\]

\[
i = 1 \quad T = \begin{array}{c} \_A \_A \_A \_A \_? \end{array} \\
P = \begin{array}{c} \_A \_A \_A \_A \_A \_B \end{array}
\]

Note - if we take the mismatching character into consideration, then we can see right away that this shift is doomed - but in our version of KMP, we’re going to omit any consideration of this character.
Questions

1. We always perform the smallest shift that does not "contradict" the portion that we have already matched. Why choose the smallest?

2. What is the general relationship between the pattern, the number of characters matched, and how many shifts we can make? (i.e. how many shifts can we rule out?)
Another Observation

We don’t need to re-examine characters that matched but did not get shifted past.

• For example:

   \[ i = 0 \quad T = \begin{array}{cccc} A & A & A & A \end{array} \times \begin{array}{c} A \end{array} \ldots \\
   P = \begin{array}{cccc} A & A & A & A \end{array} B \]

   \[ i = 1 \quad T = \begin{array}{cccc} A & A & A \end{array} \times \begin{array}{c} A \end{array} \ldots \\
   P = \begin{array}{cccc} A & A & A \end{array} A B \]

   We can only shift by 1...

   But we don’t need to check these because we’ve proven they must match

   Start checking here...
The Knuth-Morris-Pratt Algorithm

1. Set $i = 0$ and $j = 0$. (As before, $i$ will be the shift - the index in T that we are currently testing. $j$ will be the index in P).

2. Match as in the brute-force algorithm, but when a mismatch is detected:

   - If $j == 0$, increment $i$.
   - Otherwise, let $len$ be the length of the longest suffix of the first $j$ characters in the pattern that appears as a prefix of the first $j$ characters in the pattern. Let the $skip$ be $j - len$.

   Increment $i$ by the skip, and decrement $j$ by the skip.

Notice that the amount of the skip depends only on the pattern, not on the text! We can precompute a table of skips for each prefix of $P$ ahead of time, so that in the second step of the algorithm, determining the skip is simply a table lookup.
Determining the Knuth-Morris-Pratt skipArray for the string "ababaca"

- The 0th element of skipArray should never be used - we set it to 0. If we have a mismatch on the first character we compare, we just increment i by one and continue the search.

- The 1th element of skipArray is set to 1. If we have a mismatch on the 2nd character we look at, we should increment i by one, and start comparing at the start of P again (decrementing j by 1).

- skipArray contains:

```
skipArray:  0 1
```
skipArray Example (continued)

Now things start to get interesting... We need to fill in skipArray[2]. (This would be used if we matched "ab", but we failed to match the next ’a’.)

- We want to find the longest suffix of "ab" (not including "ab" itself) that’s also a prefix of "ab".

  a
  b

  a
  b

- We see that there is no such suffix. We set skipArray[2] to be the difference between the length of what we have matched ("ab" - length 2) and the length of the prefix/suffix.

- Now skipArray contains:

  skipArray: 0 1 2

- We know that if we match the first and second letters, but not the third, we can increment i by two, and start comparing at the start of P (decrementing j by 2).
skipArray Example (continued)

Now we need to fill in skipArray[3]. (This would be used if we matched "aba", but we failed to match the next 'b'.)

- We want to find the longest suffix of "aba" (not including "aba" itself) that's also a prefix of "aba".

```
   a b a
   a b a
```

- We see that there is one such suffix, "a". We set skipArray[3] to be the difference between the length of what we have matched ("aba" - length 3) and the length of the prefix/suffix ("a" - length 1).

- Now skipArray contains:

```
skipArray:  0 1 2 2
```

- We know that if we match the first, second and third letters, but not the fourth, we can increment i by two, and start comparing at the second letter of P (decrementing j by 2).
skipArray Example (continued)

Filling in skipArray[4] and skipArray[5] works in a similar manner so that skipArray will contain:

\[
\text{skipArray: } 0 \quad 1 \quad 2 \quad 2 \quad 2 \quad 2
\]

Filling in skipArray[6] is more interesting. (This would be used if we matched "ababac", but we failed to match the last 'a'.)

- We want to find the longest suffix of "ababac" that's also a prefix of "ababac".

```
ababac
  ababac
    ababac
      ababac
        ababac
          ababac
            ababac
```

skipArray Example (continued)

• We see that there is no such suffix. We set skipArray[6] to be the difference between the length of what we have matched ("ababac" - length 6) and the length of the prefix/suffix.

• Now skipArray contains:

\[
\text{skipArray:} \begin{array}{cccccc}
0 & 1 & 2 & 2 & 2 & 6 \\
\end{array}
\]

• We know that if we match the first six letters of this string, but not the last, we can increment i by 6, and start comparing at the first letter of P (decrementing j by 6).
Another Observation

• The Knuth-Morris-Pratt algorithm we’ve shown will do more work than necessary when searching for patterns like "A A A A A X" in text like "A A A A A ... A A".

• We can do better...

• Pathological Case:

\[
T = \text{A A } \text{X} \text{A A X A A ...}
\]

\[
P = \text{A A A}
\]

The pattern contains no X’s. If we look at this character first, we can save a lot of work by shifting all the way past this X.

In this manner, we can completely avoid looking at some of the text characters! This is the idea behind the Boyer-Moore string searching algorithm.
The Boyer-Moore Algorithm

Similar to KMP, but starts at the end of the pattern:

\[
\begin{align*}
i & = 0 & T &= A A A A X Y Z \\
j & = M-1 & P &= A B A A \\
i & = 0 & T &= A A A A X Y Z \\
j & = M-2 & P &= A B A A \\
i & = 0 & T &= A A A A X Y Z \\
j & = M-3 & P &= A B A A \\
\end{align*}
\]

So how far do we skip?

Again, the skip is the minimum distance that we know cannot contain a match, based on our knowledge of the pattern and the part of the string we’ve seen/matched.

- We know that the pattern requires a ’B’ in order to match, but the next two characters are ’A’s. So we shift by at least 3 positions.
- We also know (since we know the pattern) that the first character of P is an ’A’, so we can shift by 3 positions.
- If the potential shift was smaller, then we could take into account the mismatching character as well.
How Far to Skip: Two Heuristics

1. The Bad-Character Heuristic.

\[ d \text{ is the number of characters that the pattern can shift to the right that guarantees that the mismatched text character } (T[i+j]) \text{ will match the rightmost occurrence of the mismatched character in the pattern.} \]

The mismatching character matches a character earlier in the pattern.

\[ i = k \]
\[ j = M-3 \]

\[ T = \ldots C \overbrace{A A} \ldots \]
\[ P = A C \underbrace{A A} \]

**Shift by 1**

\[ i = k+1 \]
\[ j = M-1 \]

\[ T = \ldots C \overbrace{A A} \ldots \]
\[ P = A C \underbrace{A A} \]

*If the mismatched character doesn’t occur in the pattern, then the pattern may be moved completely past the mismatched character in the text.*

*If the rightmost occurrence of the mismatched character in the pattern is to the right of the current mismatched character position, then this heuristic makes no proposal.*
Two Heuristics (continued)

2. The Good-Suffix Heuristic

\[ d \text{ is the smallest shift that guarantees that any pattern characters that align with the good suffix previously found in the text will again match those suffix characters.} \]

• Example #1:

\[
\begin{align*}
&i = k \\
&j = M-3 \\
&T = \ldots C A A \ldots \\
&P = A A B A A
\end{align*}
\]

\[
\begin{align*}
&i = k+3 \\
&j = M-3 \\
&T = \ldots C A A \ldots \\
&P = A A B A A
\end{align*}
\]

We have matched two last two characters of the pattern and failed to match the third.

Shifting by one or two will put the B (which we have seen) under an A (we know this from the fact that we matched two A’s).

So, we must shift the B past the end of the match, but we should not shift more, because we want to leave the "A A" lined up.
The Good-Suffix Heuristic (continued)

• Example #2

\[ i = k \quad T = \quad \ldots C A A \ldots \]
\[ j = M-3 \quad P = \quad B B B A A \]

\[ i = k+3 \quad T = \quad \ldots C A A \ldots \]
\[ j = M-3 \quad P = \quad B B B A A \]

Once again, we have matched two characters and then failed to match the third.

Like before, we know that we cannot shift the B to a position under the two A’s we have matched.

But we need to push it even farther to the right because *no* part of the remaining part of the pattern matches "A A".
The Boyer-Moore Algorithm

Let \( i = 0 \) and \( j = 0 \). Loop until the end of the text is reached (when \( i == N-M \)) or a match is found:

1. Attempt to match the pattern to the text at shift \( i \), starting from the end of the pattern \( (j = M - 1) \) and moving backward toward the start. If the pattern completely matches the text at this shift, return \( i \). Otherwise, let \( j \) be the index in the pattern where the mismatch occurred. (The number of successfully matched characters is therefore \( M - j \)).

2. Advance \( i \) by \( d \) positions. The skip \( d \) is chosen to be the larger \( d \) that results from each of the two heuristics:

   The Bad-Character Heuristic

   The Good-Suffix Heuristic

As with Knuth-Morris-Pratt, we can precompute the skips that result from the heuristics, because the skips depend only on the pattern, not on the text.
Generating the Boyer-Moore skipArray

• Instead of using a 1-dimensional table, as we did with Knuth-Morris-Pratt, we use a 2-dimensional table - one dimension contains the string (each character a column), and the other dimension contains all the characters in the alphabet (so we can take into account the value of the mismatched character).

• Example: P = "toot"

<table>
<thead>
<tr>
<th></th>
<th>t (0)</th>
<th>o (1)</th>
<th>o (2)</th>
<th>t (3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t (0)</td>
<td>--</td>
<td>3</td>
<td>3</td>
<td>--</td>
</tr>
<tr>
<td>o (1)</td>
<td>3</td>
<td>--</td>
<td>--</td>
<td>1</td>
</tr>
<tr>
<td>other (2)</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

• BMskipArray [0][0], [0][3], [1][1], [1][2] should never be used.

• BMskipArray[0][2]

\[
\begin{align*}
T &= \ldots tt \ldots \\
P &= t o ot
\end{align*}
\]

Bad Character Heuristic: --
Good Suffix Heuristic: skip 3
skipArray Example (continued)

- BMskipArray[1][3]

\[
\begin{align*}
T &= \ldots 0 \ldots \\
P &= t o o t \\
\end{align*}
\]

Bad Character Heuristic: skip 1
Good Suffix Heuristic: --

- BMskipArray[2][0]

\[
\begin{align*}
T &= \ldots x o o t \ldots \\
P &= t o o t \\
\end{align*}
\]

Bad Character Heuristic: skip 1
Good Suffix Heuristic: skip 3

- BMskipArray[2][3]

\[
\begin{align*}
T &= \ldots x \ldots \\
P &= t o o t \\
\end{align*}
\]

Bad Character Heuristic: skip 4
Good Suffix Heuristic: --
Analysis of Boyer-Moore

• The code shown in the coursebook for precomputing the skip array in $O(M^3)$. It is possible to compute the array in $O(M)$.

• During the actual search, each character in the text is visited \textit{at most} once. (Some characters of the text may be skipped completely.) This is $O(N)$. 
Rabin-Karp String Searching

Observation:

• If two strings hash to the same value then they \textit{might} be identical. (Depending on our faith in our hash function, we can change "might be" to "probably are", as in Weiss 5.9)

• If two strings hash to different values (using the same hash function), then they \textit{cannot} be identical.

Algorithm:

1. Let hash(P) be the hash of the pattern.

2. At each shift (i = 0 through i = N-M):

   • Compute hash(T[i]) = the hash of the substring of length M starting at shift i in T.
   
   • If hash(T[i]) == hash(P), perform an explicit comparison. If there is a match, return i.

3. If the end of the text is reached without finding a match, return -1.
Analysis of Rabin Karp

• As stated, this has the potential to be a terrible algorithm.

Computing the hash of a string of length $M$ is $O(M)$. (In fact, if we really want to be pessimistic, the most we can really say is that it should be $\Omega(M)$ because we know we should consider each character, and whatever we do with each character can’t be quicker than $O(1)$ - and it might take longer.)

• Therefore, it appears that at best this method is as bad as the brute-force algorithm is at worst!

• Although this method looks awful, perhaps we can find a hash function that has properties we can exploit...
Hashing Related String

The key observation is this: consider the following text, and a pattern of length 5:

\[ T = A \ B \ A \ C \ B \ A \ C \ B \ A \ B \ D \]

We’ll compute

hash (A \ B \ A \ C \ B)

hash (B \ A \ C \ B \ A)

hash (A \ C \ B \ A \ C)

hash (C \ B \ A \ C \ B)

hash (B \ A \ C \ B \ A)

hash (A \ C \ B \ A \ B)

hash (C \ B \ A \ B \ D)

- Each string we hash is similar to the previous, except that it is "shifted" by one position - the first character is gone, the next M-1 characters are shifted left, and a new character appears in position M-1.

- Perhaps we can find a hash function that allows us to make use of this...
The Fast Hash Function

We want to find a hash function that computes to a hash value in the range 0..\(p-1\).

Let \(S\) be the string, with length \(L\).

- Using C notation, let \(S[i]\) denote the \(i^{\text{th}}\) character in string \(S\).

- Each character from the alphabet will be assigned a unique number in the range 0 to \(b-1\), where \(b\) is the base (the number of characters in the alphabet).

For example, 8-bit bytes represent characters on our machines, and have numeric values 0 through \(2^8-1\) (255).

\[
hash(S) = \left( \sum_{i=0}^{L-1} S[(L-1-i)b^i] \right) \mod p
\]

\[
= (S[0]b^{L-1} + S[1]b^{L-2} + ... + S[L-1]b^0) \mod p
\]
Computing the Hash of Subsequent Strings

- Let $S_i$ be the substring of $T$ of length $M$ starting at shift $i$.

  \[
  \text{hash} (S_i) = (T[i]b^{M-1} + T[i+1]b^{M-2} + ... + T[i+M-1]) \mod p
  \]

  \[
  \text{hash} (S_{i+1}) = (T[i+1]b^{M-1} + T[i+2]b^{M-2} + ... + T[i+M]) \mod p
  \]

- After doing a little algebra:

  \[
  \text{hash} (S_{i+1}) = \left( b \cdot \text{hash} (S_i) - T[i]b^M + T[i+M] \right) \mod p
  \]

- So, if we have hash($S_i$), there are four simple steps to get hash($S_{i+1}$):
  
  1. Multiply by $b$
  2. Subtract $T[i]b^M$
  3. Add $T[i+M]$
  4. Take the modulus (mod $p$)

  *We do not have to rehash the whole string.*

- We can take the modulus early and often, so that all of these calculations involve small, easy-to-use numbers.
Analysis of Rabin-Karp

• Rabin-Karp visits each character in the text at least twice (more if there is a potential match), once to shift the character in, and once to shift it out.

• The amount of work is unrelated to M at all, except when the hash values match but the pattern doesn’t match the subset of the text.

• It can be made quite speedy (see the coursebook) by choosing the right modulus and the right $b$.

Special-purpose hardware can be constructed to implement the Rabin-Karp string searching algorithm, because it doesn’t depend on the pattern.

• It is easily generalized to multi-dimensional searching - "does this array appear as a subarray in that array?" - unlike the other algorithms we’ve looked at.