Students should be familiar with inductive proofs, recursion, data structures, and programming at the CS3114 level.

String Matching

Let \( A = a_1 a_2 \cdots a_n \) and \( B = b_1 b_2 \cdots b_m \), \( m \leq n \), be two strings of characters.

**Problem:** Given two strings \( A \) and \( B \), find the first occurrence (if any) of \( B \) in \( A \).

- Find the smallest \( k \) such that, for all \( i, 1 \leq i \leq m \), \( a_{k+i} = b_i \).

**String Matching Example**

\[
\begin{align*}
A & = \text{xyxxyxyxyyxyxyxyyxyxyxx} \\
B & = \text{xyxyyxyxyxx}
\end{align*}
\]

1: \( \text{xy} \)    
2: \( x \)    
3: \( xy \)    
4: \( xyxy \)    
5: \( x \)    
6: \( xyxyxyxyxxyxx \)    
7: \( x \)    
8: \( xyx \)    
9: \( x \)    
10: \( x \)    
11: \( xyxyy \)    
12: \( x \)    
13: \( xxyxyxyxyxyxxyx \)

\( O(mn) \) comparisons in worst case.

**String Matching Worst Case**

Brute force isn’t too bad for small patterns and large alphabets.

However, try finding: \( \text{yyyyyxy} \)

\[
\text{in: yyyyyyyyyyyyyyyyy}
\]

Alternatively, consider searching for: \( \text{xyyyyy} \)

Our example was a little pessimistic… but it wasn’t worst case!

In the second example, we can quickly reject a position - no backtracking.
Finding a Better Algorithm

Find \( B = \text{xyxyxyxyxx} \) in \( A = \text{xyxxxyxyxyxyxyxyxyxx} \).

When things go wrong, focus on what the prefix might be.

\[
\begin{align*}
\text{xy}xy & \quad \text{no chance for prefix until third } x \\
\text{xyxy} & \quad \text{xyx could be prefix} \\
\text{xyxyxy} & \quad \text{last } \text{xyx} \text{ could be prefix} \\
\text{xyxyxyxyxx} & \quad \text{success!}
\end{align*}
\]

Knuth-Morris-Pratt Algorithm

- **Key to success:**
  - Preprocess \( B \) to create a table of information on how far to slide \( B \) when a mismatch is encountered.
  - **Notation:** \( B(i) \) is the first \( i \) characters of \( B \).
  - For each character:
    - We need the maximum suffix of \( B(i) \) that is equal to a prefix of \( B \).
  - \( \text{next}(i) = \) the maximum \( j \) \((0 < j < i - 1)\) such that \( b_{i-j}b_{i-j+1} \cdots b_{i-1} = B(j) \), and 0 if no such \( j \) exists.
  - We define \( \text{next}(1) = -1 \) to distinguish it.
  - \( \text{next}(2) = 0 \). Why?

Computing the table

\[
B = \begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\text{x} & \text{y} & \text{x} & \text{y} & \text{y} & \text{x} & \text{y} & \text{x} & \text{x} \\
-1 & 0 & 0 & 1 & 2 & 0 & 1 & 2 & 3 & 4 & 3
\end{array}
\]

- The third line is the “next” table.
- At each position ask “If I fail here, how many letters before me are good?”

How to Compute Table?

- By induction.
- **Base cases:** \( \text{next}(1) \) and \( \text{next}(2) \) already determined.
- **Induction Hypothesis:** Values have been computed up to \( \text{next}(i - 1) \).
- **Induction Step:** For \( \text{next}(i) \): at most \( \text{next}(i - 1) + 1 \).
  - When? \( b_{i-1} = b_{\text{next}(i-1)+1} \).
  - That is, largest suffix can be extended by \( b_{i-1} \).
  - If \( b_{i-1} \neq b_{\text{next}(i-1)+1} \), then need new suffix.
  - But, this is just a mismatch, so use \( \text{next} \) table to compute where to check.

Induction step: Each step can only improve by 1.

While this is complex to understand, it is efficient to implement.
Complexity of KMP Algorithm

- A character of $A$ may be compared against many characters of $B$.
  - For every mismatch, we have to look at another position in the table.
- How many backtracks are possible?
- If mismatch at $b_k$, then only $k$ mismatches are possible.
- But, for each mismatch, we had to go forward a character to get to $b_k$.
- Since there are always $n$ forward moves, the total cost is $O(n)$.

Example Using Table

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>x</td>
<td>y</td>
<td>x</td>
<td>y</td>
<td>y</td>
<td>x</td>
<td>y</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

A: $x y x y x y y x y x y x y x$

- $x y x$ next(4) = 1, compare B(2) to this
- $-x y$ next(2) = 0, compare B(1) to this
- $x y y y$ next(5) = 2, compare to B(3)
- $-x-y x y y x y x x$ next(11) = 3
- $-x-y x y y x y x x$

Note: $-x$ means don’t actually compute on that character.

Boyer-Moore String Match Algorithm

- Similar to KMP algorithm
- Start scanning $B$ from end of $B$.
- When we get a mismatch, we can shift the pattern to the right until that character is seen again.
- Ex: If “Z” is not in $B$, can move $m$ steps to right when encountering “Z”.
- If “Z” in $B$ at position $i$, move $m - i$ steps to the right.
- This algorithm might make less than $n$ comparisons.
- Example: Find $abc$ in $xbycabc$
  - $abc$
  - $abc$

Probabilistic Algorithms

All algorithms discussed so far are deterministic.

Probabilistic algorithms include steps that are affected by random events.

Example: Pick one number in the upper half of the values in a set.
- Pick maximum: $n - 1$ comparisons.
- Pick maximum from just over 1/2 of the elements: $n/2$ comparisons.

Can we do better? Not if we want a guarantee.
Probabilistic Algorithm

- Pick 2 numbers and choose the greater.
- This will be in the upper half with probability 3/4.
- Not good enough? Pick more numbers!
- For \( k \) numbers, greatest is in upper half with probability \( 1 - 2^{-k} \).
- Monte Carlo Algorithm: Good running time, result not guaranteed.
- Las Vegas Algorithm: Result guaranteed, but not the running time.

Monte Carlo Algorithm: Good running time, result not guaranteed.
Las Vegas Algorithm: Result guaranteed, but not the running time.

Searching Linked Lists

Assume the list is sorted, but is stored in a linked list.

Can we use binary search?
- Comparisons?
- “Work?”

What if we add additional pointers?

“Perfect” Skip List

What is the access time? \( \log n \).
We can insert/delete in \( \log n \) time as well.

Building a Skip List

Pick the node size at random (from a suitable probability distribution).
Skip List Analysis (1)

What distribution do we want for the node depths?

```c
int randomLevel(void) { // Exponential distribution
    for (int level=0; Random(2) == 0; level++);
    return level;
}
```

What is the worst cost to search in the “perfect” Skip List?

What is the average cost to search in the “perfect” Skip List?

What is the cost to insert?

What is the average cost in the “typical” Skip List?

Exponential decay. 1 link half of the time, 2 links one quarter, 3 links one eighth, and so on.

\[ \log n \]

Close to \( \log n \).

\[ \log n \]

\[ \log n \]

Skip List Analysis (2)

How does this differ from a BST?

- Simpler or more complex?
- More or less efficient?
- Which relies on data distribution, which on basic laws of probability?

About the same.

On average, about the same if data are well distributed.

BST relies on data distribution, while skiplist merely relies on chance.