CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2014

Title page

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 $i, 1 \leq i \leq m$, $a_{k+i} = b_i$.

no notes

String Matching Example

$A =$ `xyxxyxyxyyxyxyxyyxyxyxx` $B =$ `xyxyyxyxyxx`

```
     x y x x y x y x y y x y x y x y y x y x y x x
 1:  x y x y
 2:    x
 3:      x y
 4:        x y x y y
 5:          x
 6:            x y x y y x y x y x x
 7:              x
 8:                x y x
 9:                  x
10:                    x
11:                      x y x y y
12:                        x
13:                          x y x y y x y x y x x
```

$O(mn)$ comparisons.

$O(mn)$ comparisons in worst case.

String Matching Worst Case

Brute force isn't too bad for small patterns and large alphabets.
However, try finding: `yyyyyx`
in: `yyyyyyyyyyyyyyyx`

Alternatively, consider searching for: `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 $B =$ `xyxyyxyxyxx` in

$A =$ `xyxxyxyxyyxyxyxyyxyxyxx`
When things go wrong, focus on what the prefix might be.

```
xyxxyxyxyyxyxyxyyxyxyxx
xyxy  -- no chance for prefix until third x
   xyxyy  -- xyx could be prefix
     xyxyyxyxyxx  -- last xyxy could be prefix
            xyxyyxyxyxx -- success!
```


Not only can we skip down several letters if we track the potential prefix, we don't need even to repeat the check of the prefix letters – just start that many characters down.

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.
- $next(i) =$ 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 $next(1) = -1$ to distinguish it.
- $next(2) = 0$. Why?

In all cases other than $B[1]$ we compare current A value to appropriate B value. The test told us there was no match at that position. If $B[1]$ does not match a character of A, that character is completely rejected. We must slide B over it.

Why? All that we know is that the 2nd letter failed to match. There is no value j such that $0 < j < i - 1$. Conceptually, compare beginning of B to current character.

Computing the table

$B =$

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

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

no notes

How to Compute Table?

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

no note

Example Using Table

```
i   1   2   3   4   5   6   7   8   9  10  11
B   x   y   x   y   y   x   y   x   y   x   x
   -1   0   0   1   2   0   1   2   3   4   3

A   x y x x y x y x y y x y x y x y y x y x y x x

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

Note: `-x` means don't actually compute on that character.

no note

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
    abc
```


Better for larger alphabets.

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.

1. Pick maximum: $n - 1$ comparisons.
2. Pick maximum from just over 1/2 of the elements: $n/2$ comparisons.

Can we do better? Not if we want a **guarantee**.

no notes

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.

Pick k big enough and the chance for failure becomes less than the chance that the machine will crash (i.e., probability of even getting an answer from a deterministic algorithm).

Rather have no answer than a wrong answer? If k is big enough, the probability of a wrong answer is less than any calamity with finite probability – with this probability independent of n.

Searching Linked Lists

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

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

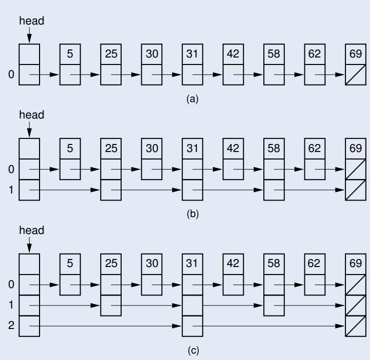
What if we add additional pointers?

Same. Is this a good model? No.

Much higher since we must move around a lot (without comparisons) to get to the same position.

Might get to desired position faster.

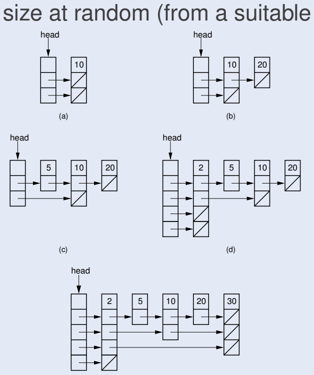
“Perfect” Skip List



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).



no notes

Skip List Analysis (1)

What distribution do we want for the node depths?

```
int randomLevel(void) { // Exponential distrib
  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.