String Matching

Let \( A = a_1a_2 \cdots a_n \) and \( B = b_1b_2 \cdots b_m \), \( m \leq n \), 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

\[
A = x\underbrace{yyxyxykxyxyxyxyy}_{2: \text{ x}}
\]

\[
B = x\underbrace{yxyxxy}_{2: \text{ x}}
\]

\[
\begin{align*}
1: & \quad x \underbrace{yxy}_{2: \text{ x}} \\
2: & \quad x \underbrace{yxy}_{2: \text{ x}} \\
3: & \quad x \underbrace{yxy}_{2: \text{ x}} . . . \\
4: & \quad x \underbrace{yxy}_{2: \text{ x}} \\
5: & \quad x
\end{align*}
\]

\[O(mn)\] 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: \( yyyyyyyyyyyyyyyyy \)

Alternatively, consider searching for: \( xyyyy \)
Finding a Better Algorithm

Find $B = xyxyxyxxyx$ in $A = xyxyxyxyxxyxxyxxyx$.

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

$xxyx$ -- no chance for prefix til last $x$

$xxyy$ -- $xxy$ could be prefix

$xxyxyxxyx$ -- last $xxy$ could be prefix

$xxyxyxxyx$ -- success!

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 $b_{i-j-1} \cdots b_{i-1} = B(j)$, and 0 if no such $j$ exists.
- We define $next(1) = -1$ to distinguish it.
- $next(2) = 0$. Why?

Computing the table

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

- 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: $next(1)$ 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 $O(n)$.

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>x</td>
<td>y</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>y</td>
<td>x</td>
</tr>
<tr>
<td>A</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>y</td>
<td>y</td>
<td>x</td>
</tr>
</tbody>
</table>

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
- $abc$
- $abc$

Order Statistics

**Definition**: Given a sequence $S = x_1, x_2, \ldots, x_n$ of elements, $x_i$ has rank $k$ in $S$ if $x_i$ is the $k$th smallest element in $S$.

- Easy to find for a sorted list.
- What if list is not sorted?
- Problem: Find the maximum element.
- Solution:
  - Problem: Find the minimum AND the maximum elements.
  - Solution: Do independently.
    - Requires $2n - 3$ comparisons.
    - Is this best?

Finding max: Compare element $n$ to the maximum of the previous $n - 1$ elements. Cost: $n - 1$ comparisons. This is optimal since you must look at every element to be sure that it is not the maximum.

We can drop the max when looking for the min. Might be more efficient to do both at once.
Min and Max

Problem: Find the minimum AND the maximum values.

Solution: By induction.

Base cases:
- 1 element: It is both min and max.
- 2 elements: One comparison decides.

Induction Hypothesis:
- Assume that we can solve for \( n - 2 \) elements.

Try to add 2 elements to the list.
- Find min and max of elements \( n - 1 \) and \( n \) (1 compare).
- Combine these two with \( n - 2 \) elements (2 compares).
- Total incremental work was 3 compares for 2 elements.

Total Work:

What happens if we extend this to its logical conclusion?

Two Largest Elements in a Set

Problem: Given a set \( S \) of \( n \) numbers, find the two largest.

- Want to minimize comparisons.
- Assume \( n \) is a power of 2.
- Solution: Divide and Conquer

Induction Hypothesis: We can find the two largest elements of \( n/2 \) elements (lists \( P \) and \( Q \)).

- Using two more comparisons, we can find the two largest of \( p_1, q_2, p_1, p_2 \).

\[
\begin{align*}
T(2n) &= 2T(n) + 2; T(2) = 1. \\
T(n) &= 3n/2 - 2.
\end{align*}
\]
- Much like finding the max and min of a set. Is this best?

A Closer Examination

- Again consider comparisons.
- If \( p_1 > q_1 \) then
  \[
  \begin{align*}
  \text{compare } p_2 & \text{ and } q_1; \quad [\text{ignore } q_2] \\
  \text{else} & \text{ compare } p_1 & \text{ and } q_2; \quad [\text{ignore } p_2]
  \end{align*}
  \]
- We need only ONE of \( p_2, q_2 \).
- Which one? It depends on \( p_1 \) and \( q_1 \).
- Approach: Delay computation of the second largest element.

Induction Hypothesis: Given a set of size \( < n \), we know how to find the maximum element and a "small" set of candidates for the second maximum element.
Algorithm

- Given set $S$ of size $n$, divide into $P$ and $Q$ of size $n/2$.
- By induction hypothesis, we know $p_i$ and $q_i$, plus a set of candidates for each second element, $C_P$ and $C_Q$.
- If $p_1 > q_1$, then
  $$new_1 = p_1; \ C_{new} = C_P \cup q_1.$$  
  Else
  $$new_1 = q_1; \ C_{new} = C_Q \cup p_1.$$  
- At end, look through set of candidates that remains.
- What is size of $C$?
- Total cost:

Lower Bound for Second Best

At least $n - 1$ values must lose at least once.
- At least $n - 1$ compares.

In addition, at least $k - 1$ values must lose to the second best.
- I.e., $k$ direct losers to the winner must be compared.

There must be at least $n + k - 2$ comparisons.

How low can we make $k$?

Adversarial Lower Bound

Call the strength of element $L[i]$ the number of elements $L[i]$ is (known to be) bigger than.

If $L[i]$ has strength $a$, and $L[j]$ has strength $b$, then the winner has strength $a + b + 1$.

What should the adversary do?
- Minimize the rate at which any element improves.
- Do this by making the stronger element always win.
- Is this legal?

Lower Bound (Cont.)

What should the algorithm do?

If $a \geq b$, then $2a \geq a + b$.
- From the algorithm’s point of view, the best outcome is that an element doubles in strength.
- This happens when $a = b$.
- All strengths begin at zero, so the winner must make at least $k$ comparisons for $2^{k - 1} < n \leq 2^k$.

Thus, there must be at least $n + \lceil \log n \rceil - 2$ comparisons.
**Kth Smallest Element**

**Problem:** Find the kth smallest element from sequence S.

(Also called selection.)

**Solution:** Find min value and discard (k times).
- If k is large, find n − k max values.

**Cost:** $O(\min(k, n - k))n$ – only better than sorting if $k$ is $O(\log n)$ or $O(n - \log n)$.

**Better Kth Smallest Algorithm**

Use quicksort, but take only one branch each time.

Average case analysis:

$$f(n) = n - 1 + \frac{1}{n} \sum_{i=1}^{n} (f(i - 1))$$

Average case cost: $O(n)$ time.

**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.

Pick $k$ big enough and the chance for failure becomes less than the chance that the machine will crash (i.e., probability of getting an answer of 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 $n$. 
Probabilistic Quicksort

Quicksort runs into trouble on highly structured input.

Solution: Randomize input order.
- Chance of worst case is then $2/n!$.

Coloring Problem

- Let $S$ be a set with $n$ elements, let $S_1, S_2, \ldots, S_r$ be a collection of distinct subsets of $S$, each containing exactly $r$ elements, $k < 2^{-r}$.
- Problem: Color each element of $S$ with one of two colors, red or blue, such that each subset $S_i$ contains at least one red and at least one blue.
- Probabilistic solution:
  - Take every element of $S$ and color it either red or blue at random.
  - This may not lead to a valid coloring, with probability $\frac{k}{2^{r-1}} \leq \frac{1}{2}$.
- If it doesn’t work, try again!

Transforming to Deterministic Alg

First, generalize the problem:
- Let $S_1, S_2, \ldots, S_r$ be distinct subsets of $S$.
- Let $s_i = |S_i|$.
- Assume $\forall i, s_i \geq 2, |S| = n$.
- Color each element of $S$ red or blue such that every $S_i$ contains a red and blue element.
- The probability of failure is at most:
  $$ F(n) = \sum_{i=1}^{k} 2^{r-S_i} $$
- If $F(n) < 1$, then there exists a coloring that solves the problem.
- Strategy: Color one element of $S$ at a time, always choosing color that gives lower probability of failure.

Deterministic Algorithm

- Let $S = \{x_1, x_2, \ldots, x_n\}$.
- Suppose we have colored $x_1, x_2, \ldots, x_{n-1}$ and we want to color $x_n$. Further, suppose $F(j)$ is an upper bound on the probability of failure.
- How could coloring $x_n$ red affect the probability of failing to color a particular set $S_i$?
- Let $P_R(i, j)$ be this probability of failure.
- Let $P(i, j)$ be the probability of failure if the remaining colors are randomly assigned.
- $P_R(i, j)$ depends on these factors:
  - whether $x_n$ is a member of $S_i$,
  - whether $S_i$ contains a blue element,
  - whether $S_i$ contains a red element.
- the number of elements in $S_i$ yet to be colored.

This principle is why, for example, the Skip List data structure has much more reliable performance than a BST. The BST’s performance depends on the input data. The Skip List’s performance depends entirely on chance. For random data, the two are essentially identical. But you can’t trust data to be random.
Deterministic Algorithm (cont)

Result:
- If \( x_i \) is not a member of \( S_i \), probability is unchanged.
  \[ P_R(i, j) = P(i, j). \]
- If \( S_i \) contains a blue element, then \( P_R(i, j) = 0. \)
- If \( S_i \) contains no blue element and some red elements, then
  \[ P_R(i, j) = 2P(i, j). \]
- If \( S_i \) contains no colored elements, then probability of failure is unchanged.
  \[ P_R(i, j) = P(i, j) \]

Key to transformation: We can calculate \( F_R \) and \( F_B \) efficiently, combined with the claim.

Deterministic Algorithm (cont)

- Similarly analyze \( P_B(i, j) \), the probability of failure for set \( S_i \) if \( x_i \) is colored blue.
- Sum the failure probabilities as follows:
  \[ F_R(j) = \sum_{i=1}^{k} P_R(i, j) \]
  \[ F_B(j) = \sum_{i=1}^{k} P_B(i, j) \]
- Claim: \( F_R(n-1) + F_B(n-1) \leq 2F(n) \).
  \[ P_R(i, j) + P_B(i, j) \leq 2P(i, j) \]

Deterministic Algorithm (cont)

- Suffices to show that \( \forall i \),
  \[ P_R(i, j) + P_B(i, j) \leq 2P(i, j) \]
- This is clear except in case (3) when \( P_R(i, j) = 2P(i, j) \).
- But, then case (2) applies on the blue side, so \( P_B(i, j) = 0 \).

Final Algorithm

For \( j = n \) downto 1 do
  calculate \( F_R(j) \) and \( F_B(j) \):
  If \( F_R(j) < F_B(j) \) then
    color \( x_j \) red
  Else
    color \( x_j \) blue.

By the claim, \( 1 \geq F(n) \geq F(n-1) \geq \cdots \geq F(1) \).
This implies that the sets are successfully colored, i.e., \( F(1) = 0 \).
Key to transformation: We can calculate \( F_R(j) \) and \( F_B(j) \) efficiently, combined with the claim.
Random Number Generators

- Most computer systems use a deterministic algorithm to select pseudorandom numbers.
- **Linear congruential method:**
  - Pick a seed \( r(1) \). Then,
  \[
  r(i) = (r(i-1) \times b) \mod t.
  \]
- Must pick good values for \( b \) and \( t \).
- Resulting numbers must be in the range: 

 Lots of “commercial” random number generators have poor performance because they don’t get the numbers right.

Must be in range 0 to \( t - 1 \).

They generate the same number, which leads to a cycle of length \(| j - i | \).

The last one depends on the start value of the seed.

Suggested generator: \( r(i) = 16807r(i-1) \mod 2^{31} - 1 \)

Random Number Generators (cont)

Some examples:

\[
\begin{align*}
  r(i) &= 6r(i-1) \mod 13 = \\
  &= \cdots 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1 \cdots \\
  r(i) &= 7r(i-1) \mod 13 = \\
  &= \cdots 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1 \cdots \\
  r(i) &= 5r(i-1) \mod 13 = \\
  &= \cdots 1, 5, 12, 8, 1 \cdots \\
  &= \cdots 2, 10, 11, 3, 2 \cdots \\
  &= \cdots 4, 7, 9, 6, 4 \cdots \\
  &= \cdots 0, 0 \cdots
\end{align*}
\]

The last one depends on the start value of the seed.

Suggested generator: \( r(i) = 16807r(i-1) \mod 2^{31} - 1 \)