Algorithms, Correctness & Efficiency – G52ACE

String Matching (guest lecture, covering for Professor Brian Logan)

Thomas Gärtner

Professor of Data Science
School of Computer Science, University of Nottingham

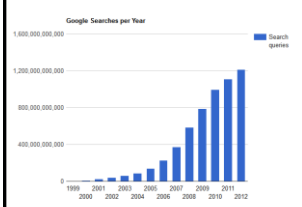
University of Nottingham
UK | CHINA | MALAYSIA

Applications & Requirements

- Spell Checkers
- Spam Filters
- Intrusion Detection Systems
- Search Engines
- Plagiarism Detection
- Bioinformatics
- Digital Forensics
- Information Retrieval Systems

Algorithms need to be super fast
- responsiveness (eg human interactions)
- coping with the amount of data (eg bio)

Google search



In 1999, it took Google one month to crawl and build an index of about 50 million pages. In 2012, the same task was accomplished in less than one minute.

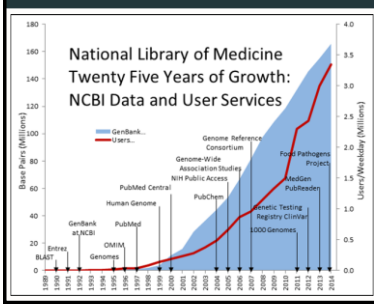
40,000 search queries every second

450 billion unique queries 2003-2012

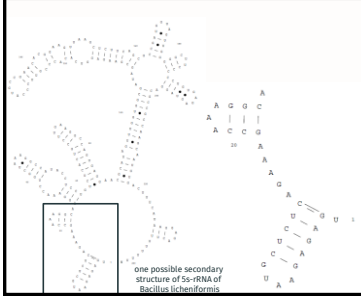
A single Google query uses 1,000 computers in 0.2 seconds to retrieve an answer.

http://www.internetlivestats.com/google-search-statistics/

Growth of data in Medicine



DNA sequencing



one possible secondary structure of 5s-rRNA of Bacillus licheniformis

Shotgun DNA sequencing results in a multiset of DNA fragments that are not ordered

A consensus sequence that is believed to represent the original DNA sequence is assembled by finding overlaps between the DNA fragments that have been sequenced

J. L. Holloway. Algorithms for String Matching with Applications in Molecular Biology. OSU 1993

Informal definition of the problem

given a
- text T
- pattern P

T and P are finite sequences of characters

determine if P occurs in T

main variants of the problem:
- simple Boolean test: return true if P occurs in T and false otherwise
- return the offset of the first occurrence of P in T
- return the offsets of all occurrences of P in T

other variants include counting, approximate matching, distance calculation, ...

Informal definition of the problem

given a
- text T
- pattern P

T and P are finite sequences of characters

determine if P occurs in T

T: ABRCADABRAC
P: ABRA

T: ABRCADABRAC ABRCADABRAC ABRCADABRAC ... ABRCADABRAC
P: ABRA ABRA ABRA ... ABRA

pattern shift pattern shift more pattern shifts 6

\o/

main variants of the problem:
- simple Boolean test: return true if P occurs in T and false otherwise
- return the offset of the first occurrence of P in T
- return the offsets of all occurrences of P in T

other variants include counting, approximate matching, distance calculation, ...

Naïve / Brute Force Algorithm

Σ: our alphabet (e.g. $\{'G','C','A','T'\}$, $['a'-'z']$, $\mathbb{N}$,...)

Σ^d: a tuple/sequence of d symbols from Σ
$P \in \Sigma^d \Rightarrow \forall\, i \in \mathbb{N}, i < d : P_i \in \Sigma$

$\Sigma^* = \cup_{d \in \mathbb{N}} \Sigma^d$: finite sequence of symbols form Σ

$|P|$: the length of a string $P \in \Sigma^*$
i.e $|P| = d$ for $P \in \Sigma^d$

$P \in [a-z]^6$
$P = (n,e,e,d,l,e)$;
$P_0 = n, P_1 = P_2 = P_5 = e, P_3 = d, P_4 = l$;
$|P| = 6$

NaiveStringMatch(T,P):
input: strings $T, P \in \Sigma^*$
output: position of P in T
or -1 if P does not occur in T

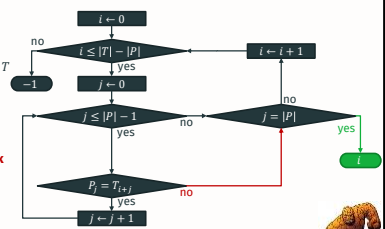
```
for i from 0 to |T| − |P|
    for j from 0 to |P| − 1
        if P_j ≠ T_{i+j} then break
    if j = |P| then return i
return −1
```

C-style interpretation of **end condition**: counter is first increased then compared

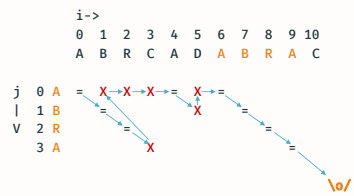
Naïve / Brute Force Algorithm

NaiveStringMatch(T,P):
input: strings $T, P \in \Sigma^*$
output: position of P in T
or -1 if P does not occur in T

```
i ← 0
while i ≤ |T| − |P|
    j ← 0
    while j ≤ |P| − 1
        if P_j ≠ T_{i+j} then break
        j ← j + 1
    if j = |P| then return i
    i ← i + 1
return −1
```



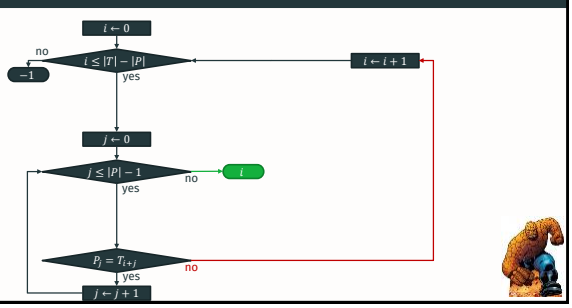
Naïve / Brute Force Algorithm



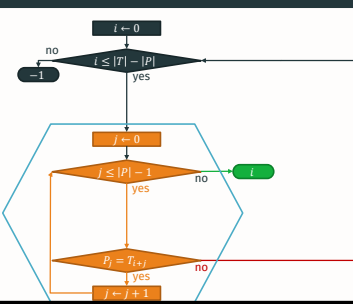
NaiveStringMatch(T,P):
input: strings T, P
output: position of P in T
or -1 if P does not occur in T

```
for i from 0 to |T| − |P|
    for j from 0 to |P| − 1
        if P_j ≠ T_{i+j} then break
    if j = |P| then return i
return −1
```

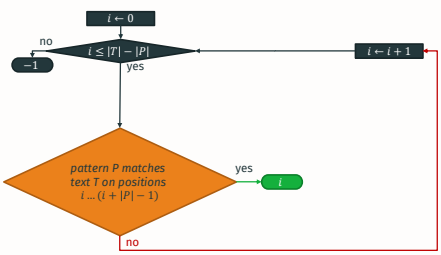
Naïve / Brute Force Algorithm



Naïve / Brute Force Algorithm: inner loop



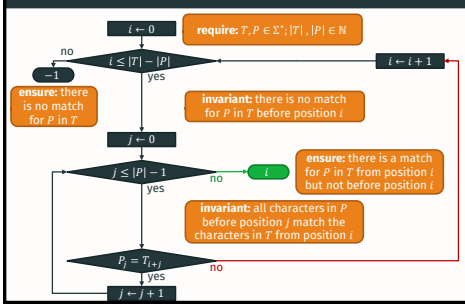
Naïve / Brute Force Algorithm: outer loop



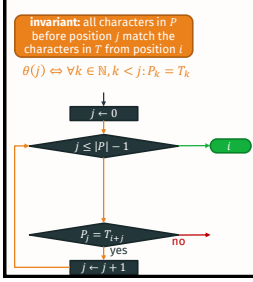
Algorithm Correctness -- Recall: Loop Invariants

- invariant must be true:
  - at the start of every iteration of the loop; and
  - when the loop terminates

Naïve / Brute Force Algorithm



inner loop correctness



initialisation:
when $j = 0$, there are no characters before position 0; so the loop invariant trivially holds prior to the first iteration of the loop

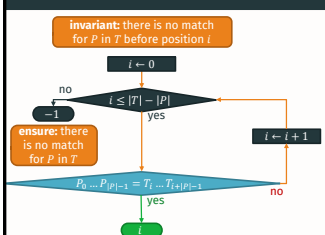
loop: consider an iteration for some j
by the loop invariant: $P_0 \ldots P_{j-1}$ matches $T_i \ldots T_{i+j-1}$

by the test, P_j matches T_{i+j}
(if not, we break and are no longer in the loop)

so afterwards, $P_0 \ldots P_j$ matches $T_i \ldots T_{i+j}$

when we increase j,
the loop invariant still holds

outer loop correctness



initialisation:
when $i = 0$, there are no characters before position 0; so the loop invariant trivially holds prior to the first iteration of the loop

loop: consider an iteration for some i
by the loop invariant, there is no match before i

by the inner loop, there is no match at i
(if there had been, we would have returned i)

so afterwards, we know there is no match at or before i

when we increase i,
the loop invariant still holds

$\theta(j) \Leftrightarrow \exists l \in \mathbb{N}, l < i : (\forall k \in \mathbb{N}, k < |P| : P_k = T_k)$

Naïve / Brute Force Algorithm Worst Case Complexity

each shift requires at most $|P|$ comparisons to validate the match

start matching at 0
the last subarray of length $|P|$ in T starts at $|T| - |P| + 1$
in the worst case we may have to check $|T| - |P| + 1$ shifts to find a match or
to determine that there isn't one
e.g., if $|T| = 10$ and $|P| = 3$ we have to check 8 shifts

NaiveStringMatch requires time $O((|T| - |P| + 1) \cdot |P|)$

T: aaaaaaaaaaaaa...a
P: aaaa...ab

the worst case running time is thus $\theta((|T| - |P| + 1) \cdot |P|)$

the bound is tight in the worst case

NaiveStringMatch(T, P):
input: strings T, P
output: position of P in T
or -1 if P does not occur in T

```
for i from 0 to |T| − |P|
    for j from 0 to |P| − 1
        if P_j ≠ T_{i+j} then break
    if j = |P| then return i
return −1
```