comp10002
Foundations of Algorithms

Semester Two, 2019

Why Algorithms?!

Lecture slides prepared by Alistair Moffat

Overview

A simple computational problem

Simply a matter of programming

Making it go faster

Adding algorithmic design

Homework

A Problem

Method 0

Method 1

Method 2

Homework

Repeated strings

A Problem

Method 0

Method 1

Method 2

Homework

Given: A sequence S of n symbols

Problem: Find all locations in S at which repeated subsequences of length m or more appear.

Example

A Problem

Method 0

Method 1

Method 2

Homework

Try

```
fat.rat.eat.bat.cat.eat.fat.rat.
```

with $m = 7$?

```
mac:  findrepeats0 7 < test1.txt
n = 32
m = 7
0 24:  fat.rat
1 25:  at.rat.
5 17:  at.eat.
count = 3
```

Just simple programming, right?

A Problem
Method 0
Method 1
Method 2
Homework

Method 0:

read the sequence S
for $i = 0$ **to** $n - m$ **do**
  for $j = i + 1$ **to** $n - m$ **do**
    compare m characters starting at $S[i]$ and $S[j]$
    if all m characters are the same **then**
      write an output line

Works just fine.

Until you give it non-trivial data.

Huh? Why so slow??

A Problem
Method 0
Method 1
Method 2
Homework

Look at those loops.

To execute the program, takes $n \times n \times m = n^2 m$ individual character comparisons.

When $n = 1{,}000$ and $m = 10$, gives $n^2 m = 10^7$ operations, so just fractions of a second.

But when $n = 1{,}000{,}000$ and $m = 100$, means $n^2 m = 10^{14}$ operations. No longer fractions of a second. No longer even seconds or minutes.

But hang on...

A Problem
Method 0
Method 1
Method 2
Homework

Why test the full m characters every time, can surely do less work *on average* than that? Leads to Method 1:

read the sequence S
for $i = 0$ **to** $n - m$ **do**
  for $j = i + 1$ **to** $n - m$ **do**
    compare at most m characters starting at $S[i]$ and $S[j]$,
      stopping as soon as any discrepancy is noted
    if no differences are in m characters **then**
      write an output line

Method 1

A Problem
Method 0
Method 1
Method 2
Homework

This makes it significantly faster.

Worst case is still $n^2 m$ operations (what kind of sequence?)

But on average, each check fails very early; now spend kn^2 operations for a sequence of n characters, where $k \ll m$ is a small constant.

Faster, but still not fast enough to be "scalable".

For $n = 1{,}000{,}000$ gives rise to $10^{12}+$ operations. And that will still be a thousand or more seconds.

Add algorithmics

Algorithms are the aspect of computing that require science.

There will be a variety of ways of achieving any given goal, with subtle and not-so-subtle differences between them.

It is often possible to trade memory space for execution time; and exploit selective pre-computation.

Sophisticated data structures are chosen to match the exact mix of operations that need to be supported.

A Problem

Method 0

Method 1

Method 2

Homework

Adding a suffix array

read the sequence S
for $i = 0$ **to** $n - m$ **do**
    $P[i] \leftarrow i$
sort $P[i]$ based on $S[P[i]], S[P[i]+1], \ldots$
for $i = 0$ **to** $n - m$ **do**
    compare at most m characters starting at
        $S[P[i]]$ and $S[P[i+1]]$
    if no differences are found **then**
        write an output line

Finds one instance of each repetition; easy modification finds all of them.

Execution time?

A Problem

Method 0

Method 1

Method 2

Homework

Sorting n items takes fewer than $2n \log n$ comparisons.

In a simple implementation, each step is a string comparison over (still never more than) m characters.

So for $n = 1{,}000{,}000$ and $m = 100$ now have fewer than $\approx 2mn \log n = 4 \times 10^9$ operations.

That should take just a second or so!

One prepared earlier...

Method	0	1	2
$n = 10{,}000$	1.0 s	0.1 s	0.005 s
$n = 100{,}000$	94 s	11.2 s	0.04 s
$n = 1{,}000{,}000$		1102 s	0.47 s
$n = 10{,}000{,}000$			7.50 s
$n = 100{,}000{,}000$			113 s
$n = 1{,}000{,}000{,}000$	*can make estimates*		
	3 m	35 y	24 m

(Using $m = 25$ and W&P up to 1M, then $m = 250$ and WSJ).

Are we there yet?

Method 2 can be improved if a tailored sorting mechanism is used.

Using a ternary Quicksort (a couple of hours of implementation effort), Method 3 runs around 50% faster, and takes 50 seconds for 100 MB.

Using an explicit method for building a suffix array (perhaps a couple of days of implementation effort) might give rise to another halving.

Are we there yet??

Careful tuning and hand optimization might halve it again, to maybe 10-15 seconds for 100 MB.

And would mean that a terabyte of data can be processed in 4 days.

Are we there yet???

No, not really.

The problem now is that sorting requires random access into the underlying string, and so each processing node would have to store the whole string, plus part of the suffix array.

Ooops!

Setting up a successful parallel computation requires a deep understanding of what the computation is doing and how it can be parallelized.

A Problem

Method 0

Method 1

Method 2

Homework

Method 4

A Problem
Method 0
Method 1
Method 2
Homework

Method 4 uses more (disk) space, but all processing is sequential, making it amenable to a parallel implementation over a cluster.

Only a little more complex, and executes fast enough: with $n = 100{,}000{,}000$ and $m = 250$, requires 40 seconds; with estimated time for $n = 10^9$ of 7 minutes, and for $n = 10^{12}$ of 7 days.

Or, 10 minutes when spread over 1,024 processors.

As an invention, Method 4 is patentable and/or publishable!

One to try at home

A Problem

Method 0

Method 1

Method 2

Homework

Given: A sequence S of n symbols.

Problem: Determine if there is any value in S that occurs more than $n/2$ times, and if so, what the value is.

Target: *Worst case linear time, constant additional space.*

Computing research

A Problem

Method 0

Method 1

Method 2

Homework

As researchers, we like nothing more than:

- Real data, and a precise description of the transformation or processing that must be performed against it; plus
- An understanding of what the end application/benefit will be from the desired techniques; plus
- Smart students to work with us.

Algorithms are Fun!