Fast, faster, fastest: Algorithms in cryptography and bioinformatics

Benjamin Burton

School of Mathematics and Physics

The University of Queensland

Outline

1. The problem and its applications
2. Quadratic algorithms: Brute force
3. Log-linear algorithms: Graphical representations
4. A linear algorithm: As good as it gets
5. Further reading

The fixed density problem

A *bitstream* is a sequence of zeroes and ones:

010110101100

Its *density* = (number of ones / number of digits) $\in [0, 1]$

Problem

Given a bitstream and a ratio $\theta \in [0,1]$, what is the longest substring of density θ?

Examples:

- $\theta = 0.4 \longrightarrow$ 0101101<u>01100</u> (length 5)
- $\theta = 0.6 \longrightarrow$ 0<u>1011010110</u>0 (length 10)
- $\theta = 0.8 \longrightarrow$ no solution
- $\theta = 1.0 \longrightarrow$ 010<u>11</u>0101100 (length 2)

Applications: Cryptography

Randomness testing is important for cryptography:

- “Random” number generators produce cryptographic keys
- Stream ciphers are intended to “look” random

Any unwanted structure or predictability $\longrightarrow$ **potential attack**

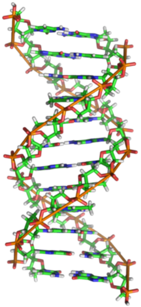
Boztaş, Puglisi and Turpin (2009):

- Developed randomness tests using the fixed density problem
- Identified potential weakness in the DRAGON stream cipher

Applications: Bioinformatics

Locating substrings with various density properties is also important for bioinformatics:

- DNA consists of T–A pairs (zero bits) and G–C pairs (one bits)
- High-density substrings $\leftrightarrow$ **GC-rich regions**
- GC-richness relates to gene density and length, recombination rates, patterns of codon usage, evolution and natural selection, and more



Potential applications also in *image processing*.

Our aim today

Assumptions:

- Given a bitstream $x_1, x_2, \ldots, x_n$ of length n
- Given a ratio $\theta = s/t \in [0, 1]$ where $0 \leq s \leq t \leq n$ and $\gcd(s, t) = 1$

Aim

To find an **algorithm** that can solve the fixed density problem in **as fast a time as possible.**

How do we measure "fast"?

- Computational complexity: $O(n^2)$, $O(n \log n)$, ...
  $\longrightarrow$ asymptotic behaviour as n grows large
- Assume that $+$, $\times$, ... are constant time operations

Quadratic algorithms: Brute force

A naïve brute force algorithm is *cubic*, i.e., $O(n^3)$:

Algorithm

procedure BRUTEFORCE($x_1, \ldots, x_n,\ \theta = s/t$)
    best $\leftarrow 0$
    for $a \leftarrow 1$ **to** n **do**
        for $b \leftarrow a$ **to** n **do**
            Count the ones in $x_a, \ldots, x_b$ ▷ This step is $O(n)$
            if density $= \theta$ **then**
                if $b - a + 1 > best$ **then**
                    $best \leftarrow b - a + 1$
    Output *best*

Outputs just the length, but easily modified to output the substring.

Quadratic algorithms: Brute force (ctd.)

A cheap trick can make this *quadratic*, i.e., $O(n^2)$:

Algorithm

```
procedure POLITEFORCE(x_1, ..., x_n, θ = s/t)
    best ← 0
    for a ← 1 to n do
        count ← 0
        for b ← a to n do
            if x_b = 1 then
                count ← count + 1
            if count/(b − a + 1) = θ then          ▷ Density of x_a, ..., x_b
                if b − a + 1 > best then
                    best ← b − a + 1
    Output best
```

There are more cheap tricks where that came from!

Quadratic algorithms: SkipMisMatch

Boztaş et al. use their SkipMisMatch algorithm:

- Applies further optimisations to brute force
- Still $O(n^2)$ in the worst case
- Improves to $O(n \log n)$ in the *expected case*

Expected case is fine for randomness testing, but perhaps not for bioinformatics or image processing.

Furthermore, performance of SkipMisMatch depends heavily on θ: $\theta \sim \frac{1}{2}$ is bad, and $\theta = \frac{1}{2}$ becomes $O(n^2)$.

$\longrightarrow$ We should aim for $O(n \log n)$ or better even in the *worst case*.

Log-linear algorithms: Graphical representations

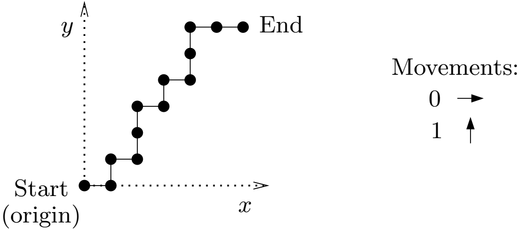
Not sure what to do? Try *drawing* the problem!

Definition

Grid representation for a bitstream:

- Start at $(0, 0)$
- Move one unit right for every 0 and one unit up for every 1

The grid representation for 010110101100:



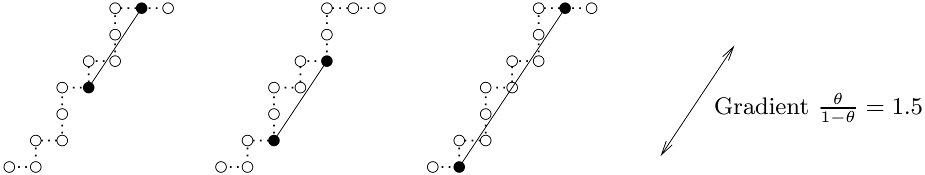
Log-linear algorithms: Graphical representations (ctd.)

How do substrings of density θ appear graphically?

Observation

A substring has density θ if and only if the line joining its start and end points in the grid representation has gradient $\frac{\theta}{1-\theta}$.

Examples in 010110101100 with density $\theta = 0.6$:



Log-linear algorithms: Working with slopes

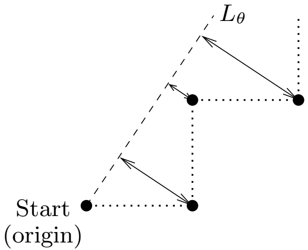
How does this help with algorithms?

- Density becomes a property of the start and end points *only*.

Observation

Draw a line L_θ through $(0,0)$ with gradient $\frac{\theta}{1-\theta}$.

*A substring has density θ if and only if the start and end points in the grid representation are the **same distance** from this line.*



Log-linear algorithms: Building an algorithm

An algorithm is becoming clear:
Compute distances, and **look for repetitions.**

We are now processing *individual points*, not pairs of points!
$\longrightarrow$ Can we escape from $O(n^2)$?

Use a *map structure* from computer science:

- Stores *key* $\mapsto$ *value* pairs
- Searching for a key is $O(\log n)$
- Inserting a new pair is $O(\log n)$

Log-linear algorithms: Building an algorithm (ctd.)

Our map will contain pairs

$$\textit{distance} \mapsto \textit{position in string}.$$

Each time we process a new point, see if the distance is already a key in our map.

- If so, we have a substring of density θ.
- If not, insert the $\textit{distance} \mapsto \textit{position}$ pair into our map.

We have $n + 1$ steps, each with time $O(\log n)$:

Theorem

Our new algorithm runs in $O(n \log n)$ time, even in the worst case!

A linear algorithm: As good as it gets

Log-linear is nice, but can we do better?

Aim for *linear*, i.e., $O(n)$. This is the *best we can possibly do*.

Our new strategy:

- Use the map-based algorithm as a starting point
- Replace the generic map with a specialised data structure, designed specifically for the task at hand

Step 1: The distance sequence

We begin by turning distances into integers.

The *distance sequence* is just distance from the line L_θ, but rescaled:

Definition

Recall that $\theta = s/t$, where $\gcd(s,t) = 1$.

For a bitstream $x_1, \ldots, x_n$, we define the *distance sequence* $d_0, d_1, \ldots, d_n$ by:

$$d_k = (t - s) \cdot (\text{\# ones in } x_1, \ldots, x_k) - s \cdot (\text{\# zeroes in } x_1, \ldots, x_k).$$

From our earlier observations, we obtain:

Lemma

A substring $x_a, \ldots, x_b$ has density θ if and only if $d_{a-1} = d_b$.

Using the distance sequence

Recall that distances are *keys* in our map.

That is, we store pairs $d_k \mapsto k$ (distances $\mapsto$ positions in the bitstream).

Our keys are now *integers*. . . but not just any integers!

Observation

Each successive key (distance) is ***always*** *obtained by adding* $+(t-s)$ *or* $-s$ *to the previous key.*

Can we use this to speed up our $O(\log n)$ map operations?

Can we "jump" from one key to the next in *constant time*, without requiring a full $O(\log n)$ search?

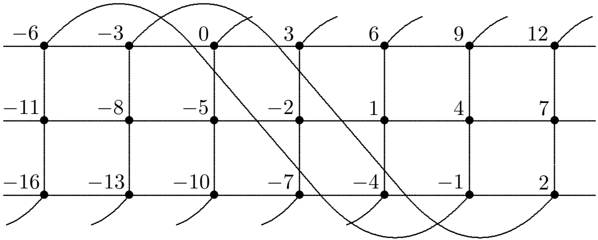
Step 2: A lattice of integers

Pull the integer number line out into a two-dimensional grid, so that both $+(t-s)$ and $-s$ are simple **local operations**.

We use infinitely many columns but only $(t-s)$ rows.

- The operation $+(t-s)$ becomes a single step to the right.
- The operation $-s$ becomes a single step down (the bottom wraps back around to the top).

For $s/t = 5/8$ we have $t - s = 3$ rows:



Using the lattice

The lattice becomes a *matrix*:

- Keys (distances) become *cells* of the matrix
- Values (positions in the bitstream) become *entries* in the matrix

We cannot store the entire matrix!

- Infinitely many cells in theory
- Still $O(n^2)$ *potential* keys in practice

However, our matrix is *sparse*:

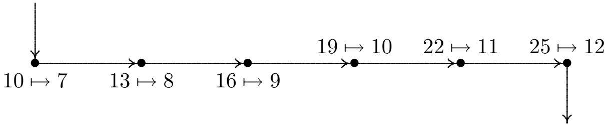
- Only $n + 1$ keys are used for any given bitstream

We can store **only the cells that we visit.**

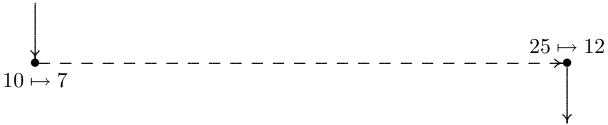
Step 3: Compressing the sparse matrix

But. . . we don't even need to store that!

A string of horizontal steps:



can be replaced by just *two points*:

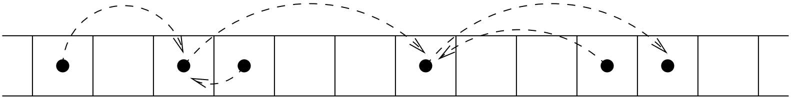


This might seem frivolous, but it turns out to be critical for achieving $O(n)$ running time.

Step 4: Pointers, pointers, pointers

So. . . how to store our sparse matrix in memory?

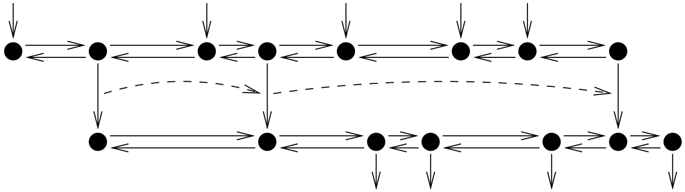
- We can't use a standard table / array, since this would be too large.
- Store our "important" cells in arbitrary memory locations, but store **pointers** in our cells that show where to find related cells.



Step 4: Pointers, pointers, pointers (ctd.)

For each cell, we store:

- Left / right pointers to adjacent "important" cells in the same row;
- A downward pointer into the next row, *only if we have travelled down from this point before*;
- For each downward pointer, we also keep a pointer to the *next downward pointer* in the same row.



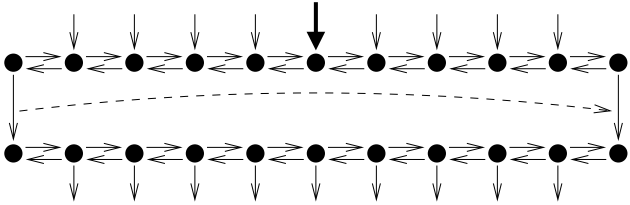
Stepping through the matrix

How do we step to the right?

- Easy—this is a local operation involving just the immediate left / right pointers.

How do we step down?

- Could be difficult—we might need to take a long walk. . .



This is definitely **not** constant time!

Dealing with the difficult case

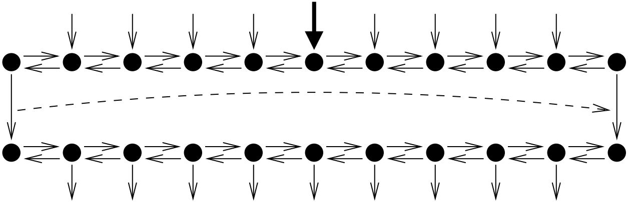
The miracle:

Theorem

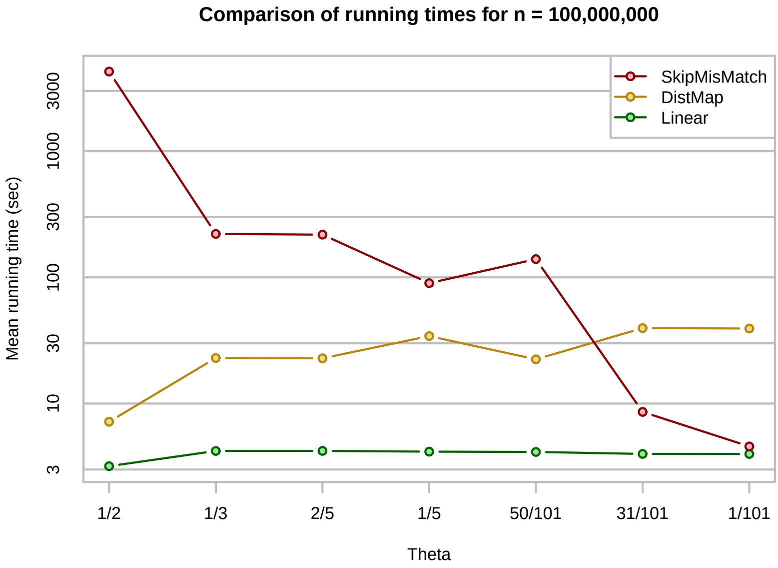
*Although a single downward step could potentially take $O(n)$ time, the **sum of all downward steps** also takes $O(n)$ time.*

*In other words, a downward step might not take constant time, but it takes **amortised constant time**.*

Essentially, we can have some slow steps but we can prove that there are so few of them that it does not matter.



But. . . does it really work?



Further Reading

Cryptographic applications for the fixed density problem:

- Serdar Boztaş, Simon J. Puglisi, and Andrew Turpin, *Testing stream ciphers by finding the longest substring of a given density*, Information Security and Privacy, Lecture Notes in Comput. Sci., vol. 5594, Springer, 2009, pp. 122–133.

This work, plus algorithms for the related *bounded density problem*:

- B.B., *Searching a bitstream for the longest substring of any given density*, `arXiv:0910.3503`, Preprint, 2009.

An excellent book on algorithms and complexity:

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, 2nd ed., MIT Press, 2001.