# Efficiency of algorithms

## Introduction

Consider the problem of finding the prime factors of

$$2^{67} - 1 = 147\,573\,952\,589\,676\,412\,927$$

This took F. N. Cole three years of Sunday afternoons at the start of the 20$^{th}$century to factor this number.

Now consider multiplying

$$761\,838\,257\,287 \times 193\,707\,721$$

This would take you a few minutes on a Sunday afternoon.

Today most e-commerce on the internet is protected by the fact that it takes years to factor large numbers (numbers with more than 300 decimal digits) even with very fast computers.

Why? The security of the RSA cryptography system is based on the fact that factoring is currently a *difficult* problem.

> **Question:** Is factoring *inherently difficult*.
> I.e., is the problem of factoring large numbers currently difficult because *there is no fast algorithm* for it, or because, although there is a fast algorithm for it, *we haven't found a fast algorithm for it yet*?

We don't currently know.

No one knows the answer to this question.

If a fast algorithm is found, it could immediately destroy the security of e-commerce.

Factoring is useful to cryptography because it is (currently) difficult to find the answer, but easy to check the answer.

> **Question:** If there is a fast algorithm to *check* an answer to a given question, does this imply that there is also a fast algorithm to *find* an answer?

We don't know the answer to this question.

- If the answer to this question is *yes*, then factoring can not be inherently difficult (bad news for e-commerce), but a large number of other problems that we *would* like to have fast algorithms for must have fast algorithms (good news).
- If the answer to this question is *no*, then a large number of problems that we would like to have fast algorithms for can not have fast algorithms. These problems are called $\mathbf{NP}$-hard problems.

This question is (roughly) the $\mathbf{P} = \mathbf{NP}$ problem.

It is one of the Clay Millennium prize questions for which there is a $1,000,000.00 (U.S.) prize.

In this part of the course we will look at:

- How do we talk about the speed of an algorithm?
- Where do we draw the line between fast and slow algorithms?

- How do we talk about the difficulty of a problem?
- Where do we draw the line between easy and difficult problems?
- What are the problem classes $\mathbf{P}$ and $\mathbf{NP}$?
- How do we show that a problem is in $\mathbf{NP}$?
- How do we show that a problem is $\mathbf{NP}$-hard? (An $\mathbf{NP}$-hard problem is one that is inherently difficult, if $\mathbf{P} \neq \mathbf{NP}$.)
- What are the practical implications of all this theory?

# The time a piece of code takes

Depends on

- Details of the machine
- How the compiler translates it.
- The “size” of the input.
- Perhaps also, the value of the input.

Example:

```
var sum := 0.0 ·
for i ∈ {0,..N} ·
    sum += A[i] ;
var average := sum/N ·
```

On any particular machine and compiler, the exact amount of time taken is

$$a \times N + b$$

where $a$ and $b$ are fixed positive coefficients. We say the algorithm is linear-time. Or that it has a time-complexity of $\Theta(N)$ (“order N”) time

Note. For large, $N$ the $b$ term becomes insignificant. For large $N$ the running time is essentially proportional to $N$.

Suppose another algorithm takes time

$$c \times N^2 + d \times N + e$$

(where $c$, $d$, and $e$ are fixed coefficients.) We say it is quadratic-time, or that it has a time complexity of $\Theta(N^2)$.

Note. For large $N$, the running time is essentially proportional to $N^2$.

Regardless of the constants, for sufficently large $N$, the linear time algorithm will be quicker.

Thm. Given a $\Theta(N)$ time algorithm and a $\Theta(N^2)$ time algorithm. There exists a size of input such that the $\Theta(N)$ algorithm is the faster one for all equal or larger sizes of inputs.

When comparing algorithms on large inputs, we can ignore:

- All but the dominating term.
- All the constants

This is good because the constants depend on the machine and the compiler and thus are hard to know and liable to change.

## Time functions of an algorithm:

For a given input $X$, the amount of time the algorithm will take: $T(X)$.

Size of input: $S(X)$, appropriate function varies from problem to problem.

Example: Sorting – $S(X)$ is the conventionally the number of items to be sorted.

Example: Array multiplication: $S(X)$ is conventionally the size of *a side of* an array.

For any number $N$, consider the set of all inputs of that size:

$$I(N) = \{X \mid S(X) = N\}$$

**Worst case time function:** For a given size of input $N$ what is the longest the algorithm will take?

$$WT(N) = \max_{X \in I(N)} T(X)$$

**Average case time function:** For a given size of input $N$ what is the expected time the algorithm will take?

Suppose $p(X)$ is the probability that an input of size $S(X)$ is $X$.

Define $AT_p(N) = \sum_{X \in I(N)} p(X) \times T(X)$

When the probability distribution is uniform we have

$$AT(N) = \frac{\sum_{X \in I(N)} T(X)}{|I(N)|}$$

# The speed of programs can be considered:

- Quantitatively – What are $T$, $WT$, $AT$.
  - Requires knowledge of: coding of algorithm as program, compiler, hardware.
  - Can be measured in tests.
- Qualitatively – what *kind* of functions are $WT$ and $AT$? Do they grow quickly or slowly?
  - Does not depend on details of coding, compiler, or hardware.
  - Does require assumption that certain operations require a fixed amount of time:
    - Usually we assume that Comparisons of ints, floats, chars. Arithmetic operations on ints, floats, chars. Fetch from and store to memory take constant time.

We will look at the qualitative speed of algorithms.

# Example: sorting algorithms

## Selection sort

```
procedure SelectionSort() (
    // Invariant: A(k,..N) is sorted
    // and ∀i, j · 0 ≤ i < k ≤ j < N ⇒ A(i) ≤ A(j)
    var k := N ·
    while k > 1 do (
        // Find maximum in A(0,..k)
            var m :=0 ·
            var big := A(m) ·
            var n := 1 ·
            // Invariant: 0 < n ≤ k and
            // A(m) = big = max{i ∈ {0, ..n} · A(i)}
            while n < k do(
                if A(n) > big then (m := n; big := A(n))
                else skip ;
                n := n+1 ) ;
        // Swap A(k-1) with A(m)
        A(k-1), A(m), k := big, A(k-1), k-1 ) )
```

Counting data comparisons: 1 per iteration of inner loop.

Iterations of inner loop. k per each iteration of outer loop.

Iterations of outer loop: N-1., with k = N-1, N-2, ..., 1

So total # of data comparisons:

$$\begin{aligned} & (N-1) + (N-2) + ... + 1 \\ = & \sum_{i=1}^{N-1} i \\ = & \ N(N-1)/2 \\ = & \ \frac{N^2 - N}{2} \end{aligned}$$

Similarly each other operation will be done a fixed, linear, or quadratic number of times.

We can write WT as

$$WT_{selection}(N) = k_2 N^2 + k_1 N + k_0$$

for some constants $k_2$, $k_1$, $k_0$, with $k_2 > 0$.

# Merge Sort:

```
void merge(int i, int j, int k, double A[N] ) {
    static double B[N] ;
    for( int r = i ; r < k ; ++r ) B[r] = A[r] ;
    for( int p=i, q=j, r=i ; p!=j || q!=k ; ) {
        if(q==k || p!=j && B[p] < B[q]) A[r++] = B[p++];
        else A[r++] = B[q++ ] ; } }

void merge_sort1(int i, int k, double A[N] ) {
    if( i < k-1 ) {
        int j = (i+k)/2 ;
        merge_sort1( i, j, A ) ;
        merge_sort1( j, k, A ) ;
        merge( i, j, k, A ) ; } }

void merge_sort( double A[N] ) {
    merge_sort( 0, N, A ) ; }
```

How it works:

We

- split the array in half.
- sort each half
- merge the two halves together.

Sorting the two halves is also done be merge sort.

(By the way, this code can be optimized to eliminate about half the copies.)

### How fast is it?

To merge two segments of total size $M$ together takes 2M data moves.

Consider the tree of calls.

For each call, the amount of work done is $c_0 + c_1(k-i)$ if $(k-i) \geq 2$ and $c_2$ when $(k-i) < 2$

For $N = 64$, there is

- One call with $(k-i) = 64$
- Two calls with $(k-i) = 32$
- ...
- 64 calls with $(k-i) = 1$

Total time for $N = 2^n$ is $c_2 N + \sum_{m=1}^{n} 2^{n-m}(c_0 + c_1 2^m)$

$$
\begin{aligned}
& c_2 N + \sum_{m=1}^{n} 2^{n-m}(c_0 + c_1 2^m) \\
= & \\
& c_2 N + c_0 \sum_{m=1}^{n} 2^{n-m} + c_1 \sum_{m=1}^{n} 2^{n-m} 2^m \\
= & \\
& c_2 N + c_0 \sum_{m=0}^{n-1} 2^m + c_1 \sum_{m=1}^{n} N \\
= & \\
& c_2 N + c_0 \left(2^n - 1\right) + c_1 N n \\
= & \\
& c_2 N + c_0 N - c_0 + c_1 N \log_2 N
\end{aligned}
$$

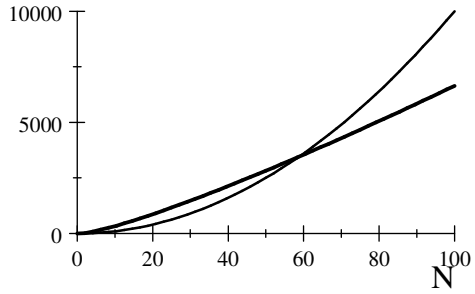
So the time is

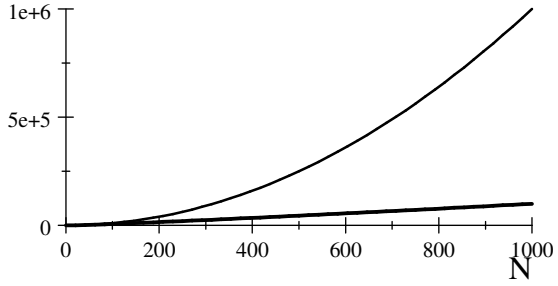
$$WT_{merge}(N) = cN \log_2 N + bN + a$$

for constants $a$, $b$, and $c$.

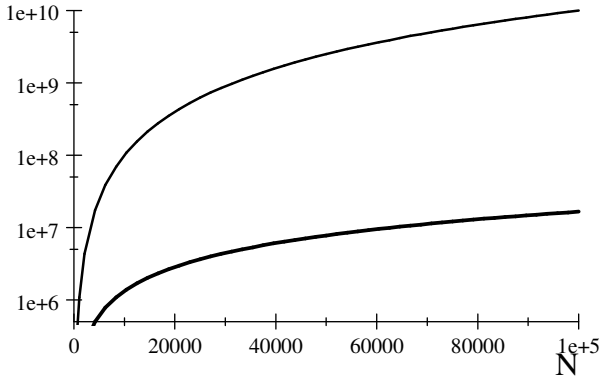
# *Which is better?*

Regardless of the constants, we can always find an input size after which $WT_{merge}$ is smaller than $WT_{selection}$.




$f(N) = N^2$ (thin) $g(N) = 10N \log_2 N$ (thick)



# The BIG-Theta and Big-Oh notation — informally

Informally:

- If $g(N) = N$ then $\Theta(g)$ is the set of functions that grow linearly.
- If $g(N) = N^2$ then $\Theta(g)$ is the set of functions that grow quadratically.
- If $g(N) = 2^N$ then $\Theta(g)$ is the set of functions that double as their argument increases.

# The BIG-Theta notation — formally

**Defn:** A real-to-real function is a function $f$ such that $f(x)$ is real, for all real $x$.

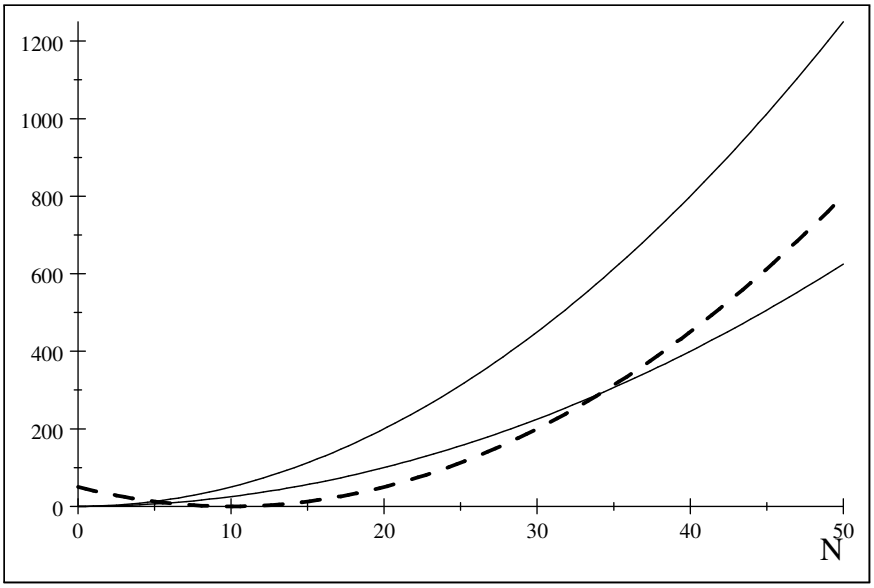
**Defn:** If $g$ is a real-to-real function then $\Theta(g)$ is a set of functions.

A function $f$ is in $\Theta(g)$ iff

- there exist positive numbers $c$, $d$, and $M$ such that
  $0 \leq c \times g(N) \leq f(N) \leq d \times g(N)$, for all $N \geq M$.

This means that, for sufficiently large $N$, $f(N)$ is trapped between two multiples of $g(N)$.

In a picture




Here $f(N) = \frac{1}{2}N^2 - 10N + 50$ and $g(N) = N^2$.

We use $c = \frac{1}{4}$, $d = \frac{1}{2}$ and $M = 40$.

The solid lines in the figure are $\frac{1}{4}N^2$ and $\frac{1}{2}N^2$.

**Exercise**: prove that for all $N \geq 40$

$$\frac{1}{4}N^2 \leq f(N) \leq \frac{1}{2}N^2$$

This shows that $f \in \Theta(g)$ where $g(N) = N^2$.

**Notation:** Usually instead of $\Theta(g)$, we actually write $\Theta(g(N))$. This is a conventional 'abuse of notation'.

For example, when you see $\Theta(N^2)$, what is really meant is $\Theta(g)$ where $g(N) = N^2$.

**Example**: We can restate the last result as $f \in \Theta(N^2)$.

**Note**:The meaning of $\Theta(g(N))$ does not depend on $N$; it depends on $g$.

**Notation**: Many writers actually write $f(N) \in \Theta(g(N))$ or even $f(N) = \Theta(g(N))$. These are unfortunate notations,

but you will often see them. In each case, what is meant is $f \in \Theta(g)$.

**Example**: If $f(N) = 2N^2 + N + 1$ then $f$ is in $\Theta(N^2)$

**Example**: If $f(N) = 2N^2 + N + 1$ then $f$ is not in $\Theta(N^3)$

**Example**: $WT_{merge}$ is $\Theta(N \times \log_2 N)$

**Example**: $WT_{selection}$ is in $\Theta(N^2)$

**Note** that, if $f \in \Theta(g)$ then, if it is defined, $\lim_{N\to\infty} \frac{f(N)}{g(N)}$ is neither 0 nor infinity, but rather some positive number.

**Also note** that, constants don't matter. For example

$$\Theta(N^2) = \Theta(\frac{1}{2}N^2) = \Theta(2N^2).$$

**And that** lower order terms don't matter. For example

$$\Theta(N \log_2 N) = \Theta(N \log_2 N + N) = \Theta(N \log_2 N + 4).$$

Usually we write the function inside $\Theta$ in the simplest possible terms.

## The complexity of an algorithm

**Defn**. If the worst-case time function of an algorithm is in $\Theta(g)$, we say that *the worst-case time complexity* of the algorithm is $\Theta(g)$ or, equivalently, that the algorithm *has a worst-case time complexity* of $\Theta(g)$.

(The words *worst-case* may be omitted.)

**Defn**. If the average-case time function of an algorithm is in $\Theta(g)$, we say that *the average-case time complexity* of the algorithm is $\Theta(g)$ or, equivalently, that the algorithm *has a average-case time complexity* of $\Theta(g)$.

**For example:**

- mergesort has a time complexity of $\Theta(N \log_2 N)$
- Selection sort has a time complexity of $\Theta(N^2)$.
- Quicksort has a worst-case time complexity of $\Theta(N^2)$, but an average case time complexity of $\Theta(N \log_2 N)$

# Upper bounds and big-Oh

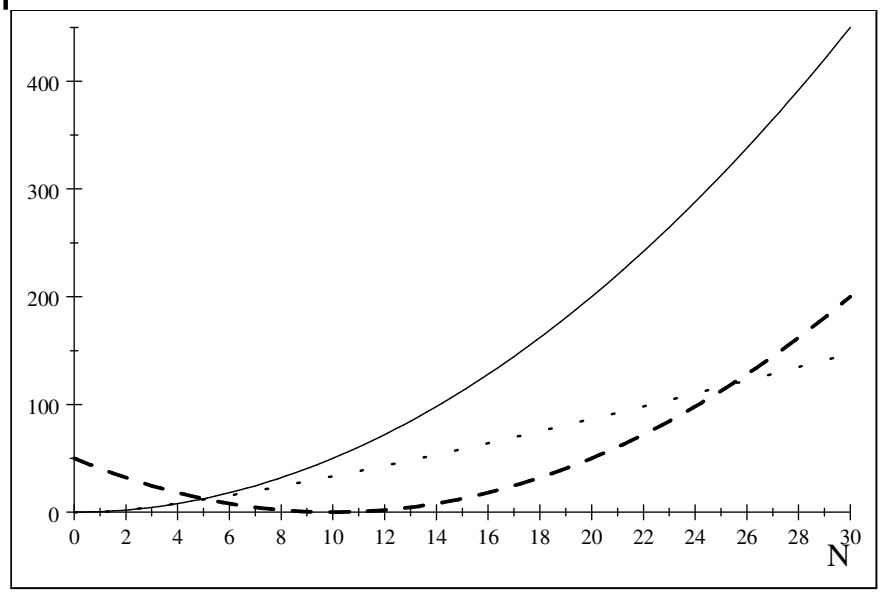
Sometimes we just want to say that there is an upper-bound on the complexity of a function.

In this case we use a set $O(g)$ of functions that do not grow significantly quicker than $g$.

Formally we define that a function $f$ is in $O(g)$ iff there exists positive constants $d$ and $M$ such that

$$0 \leq f(N) \leq d \times g(N), \text{ for all } N \geq M.$$

For example




$\frac{1}{2}N^2 - 10N + 50$ (dashed line) is in $O(N^2)$ but so is $N \log_2 N$ (dotted line).

(The solid line is $\frac{1}{2}N^2$)

Saying that the time-complexity of an algorithm is in $O(g)$ means that it will not be radically slower than an algorithm with time complexity $\Theta(g)$.

Saying that the time-complexity of an algorithm is in $O(g)$ leaves open the possibility that it could be significantly faster than an algorithm with time complexity $\Theta(g)$.

Thus $O$ sets form a hierarchy

$$\begin{aligned} O(1) \subset\ & O(\log N) \subset O(N) \subset O(N \log N) \subset O(N^2) \\ \subset\ & O(N^3) \subset \cdots \subset O(2^N) \subset O(3^N) \subset \cdots \subset O(N!) \end{aligned}$$

On the other hand, $\Theta(f)$ and $\Theta(g)$ are either equal or disjoint.

## Some rules for working with assymptotic complexity

- Ignore low order terms: if $f$ is in $O(g)$, then $\Theta(f(N) + g(N)) = \Theta(g(N))$.
  Example: $\Theta(N + \log_2 N) = \Theta(N)$
  Example: $\Theta(N^2 + N + 1) = \Theta(N^2)$
- Example: $\Theta(2N) = \Theta(N + N) = \Theta(N)$
- Ignore multiplicative constants: $\Theta(c \times f(N)) = \Theta(f(N))$
  Example $\Theta(5N^2) = \Theta(N^2)$
  Example $\Theta(\log_2 N) = \Theta(\log_2 10 \times \log_{10} N) = \Theta(\log_{10} N)$
  Convention: bases are left off logs: $\Theta(N \log_2 N) = \Theta(N \log N)$

## Finding the complexity of an algorithm

- Find an operation that is executed at least as frequently as any other operation.
- Find the number of times that operation is executed as a function of the input size.
- Find the complexity of that function.

Why? Suppose that an algorithm has $k$ operations numbered $\{0, ..k\}$.

Let $f_i(N)$ be the number of times operation $i$ is executed for the worst case input of size $N$.

Let $c_i$ be the time to execute operation $i$. Now

$$WT(N) = \sum_{i \in \{0,..k\}} c_i \times f_i(N)$$

Now let $j$ be the most frequently executed operation. I.e. $f_i \in O(f_j)$, for all $i$. Then $WT \in \Theta(f_j)$.

## Example: Searching for a string

We want to find the first occurrence of a string $p$ in a string $t$

For example

$$t = \textit{“tatatatagcttatagg”}$$
$$p = \textit{“tatag”}$$

the result is 4.

The input size will be measured by two variables:

- $N$ the length of $t$ and
- $M$ the length of $p$.

A simple algorithm is

**proc** match( **in** $p$ : **char**$^*$, **in** $t$ : **char**$^*$, **out** $found$ : **bool**,
**out** $w$ : **int** ) (
$w := 0$ ;
**var** $i := 0$ ·
// inv: $0 \le i \le \mathsf{len}(p) \ \wedge \ 0 \le w + i \le \mathsf{len}(t)$
// $\wedge\ p[0, ..i] = t[w, ..w + i]$
// $\wedge\ \neg\,(\exists v \cdot v < w \ \wedge \ t[v, ..v + \mathsf{len}(p)] = p)$
**while** $i < \ \mathsf{len}(p) \ \wedge \ w + i < \ \mathsf{len}(t)$ **do**
**if** $t(w + i) = p(i)$
**then** $i := i + 1$
**else** $w, i := w + 1, 0$ ;
$found := (i = \ \mathsf{len}(p))$ )

The invariant is illustrated by

$$t: \overbrace{\texttt{tata}}^{w}\overbrace{\texttt{tatcag}}^{t[w,..w+6]}\texttt{agcttatagg}$$
$$p: \qquad \underbrace{\texttt{tatcag}}_{p[0,..6]}\texttt{ac}$$

This algorithm checks for $p$ at each position of $t$.

Suppose the length of $p$ is $M$ and the length of $t$ is $N$.

Count comparisons.

The algorithm makes up to $M$ comparisons at each of $N - M$ places.

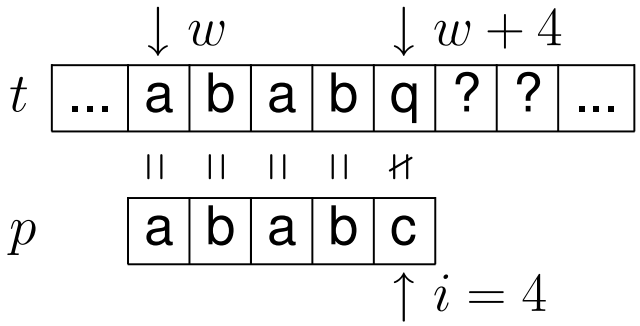
It could also make up to $(M - 1)$ comparisons at one place and $(M - 2)$ at one place, ...

So the time complexity is $\Theta(M \times N - M^2 + \sum_{i=1}^{M}(M - i)) = \Theta(M \times N)$.

# Improving the worst case

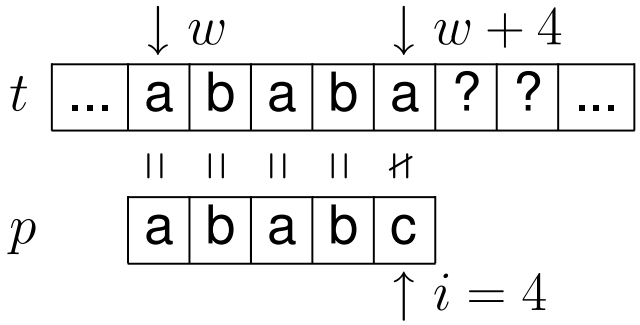
The above algorithm ‘shifts’ the pattern by one position at a time.

Consider the following mismatch:




Since ‘q’ does not appear in the first $i+1$ positions of the pattern, we can shift the pattern by $i+1$.

Consider the following mismatch:




Since the mismatch occurs at position 4 of the pattern string, we know that

$$t[w, ..w+4] = p[0, ..4]$$

We also know that $t(w+4) =$ ‘a’. Shifting the pattern by one will not work unless

$$t[w+1, ..w+5] = p[0, ..4],$$

i.e. unless

$$p[1, ..4] = p[0, ..3] \text{ and ‘a’} = p(3)$$

This is something we can determine to be false before we look at $t$. I.e. by looking only at $p$ we can determine

that, if there is a mismatch when $i = 4$ and $t(w + i) =$'a', then there is no point shifting by 1.

Likewise shifting by 2 can not work unless

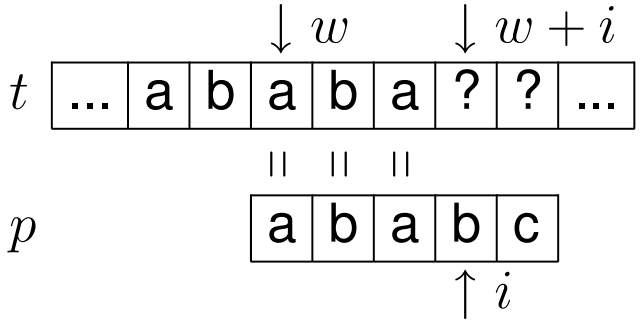
$$t[w + 2, ..w + 4] = p[0, ..2] \text{ and } t(w + 4) = p(2),$$

i.e. unless

$$p[2, ..4] = p[0, ..2] \text{ and 'a'} = p(2)$$

This is true, so shifting by 2 might work, when $i = 4$ and $t(w + i) =$'a'

After shifting by 2 we have:



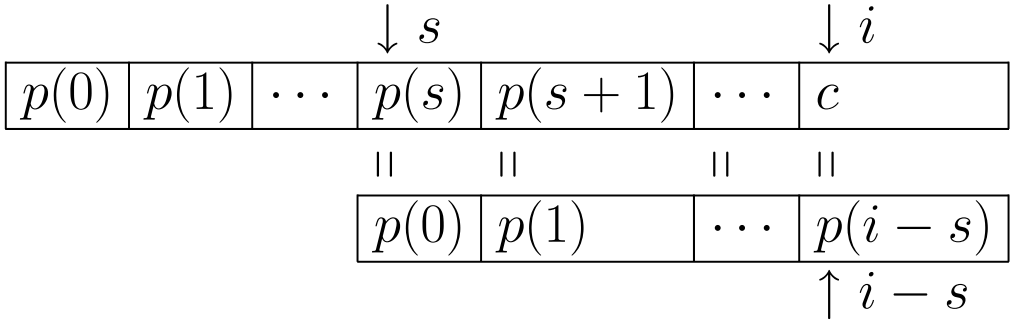

We can infer the equalities shown without further comparisons, so $i$ may be set to $3$, as shown. ($3$ is $i + 1-$ the shift)

We need never compare a target string item more than once!

Create a table indexed by:

- The number of successful comparisons: $i$ — $0 \leq i <$ len$(p)$
- The next input character: $c$.

Compute $shift(i, c)$ as the minimum value of $s$ that makes all the equalities in this picture true:




Note that:

- When $p(i) = c$, we have $shift(i, c) = 0$.
  - This means no shift.
- At the other extreme, $shift(i, c)$ is potentially as large as $i + 1$.
  - This happens, for example, when $c$ does not occur in the pattern.

$w := 0$ ;
**var** $i := 0$ ·
// inv: $0 \le i \le \mathsf{len}(p) \ \wedge \ 0 \le w + i \le \mathsf{len}(t)$
// $\wedge\ p[0, ..i] = t[w, ..w + i]$
// $\wedge\ \neg\,(\exists v \cdot v < w \ \wedge \ t[v, ..v + \mathsf{len}(p)] = p)$
**while** $i < \mathsf{len}(p) \ \wedge \ w + i < \mathsf{len}(t)$ **do** (
    **let** $s := \mathit{shift}(\, i, t(w + i)\,)$·
    $w, i := w + s, i + 1 - s$ ) ;
$\mathit{found} := (i = \ \mathsf{len}(p))$

Since, in each iteration, $w + i$ increases by one, the total number of iterations is no more than $N$, where $N$ is the length of the target string.

So the whole algorithm has time complexity of $\Theta(N)$ + the time to build the shift table. Building the shift table has complexity of $\Theta(M^2)$.

### Better yet

There is a variation on the above algorithm that uses only a table indexed by the position of the mismatch. This is the Knuth-Morris-Pratt algorithm and is $\Theta(N + M)$