Introduction to the Analysis of Complexity

Chapter 27
Are All Decidable Languages Equal?

- \((ab)^*\)
- \(WW^R = \{ww^R : w \in \{a, b\}\}^*\)
- \(WW = \{ww : w \in \{a, b\}\}^*\)
- \(SAT = \{w : w \text{ is a wff in Boolean logic and } w \text{ is satisfiable}\}\)
The Traveling Salesman Problem

Given \( n \) cities and the distances between each pair of them, find the shortest tour that returns to its starting point and visits each other city exactly once along the way.
The Traveling Salesman Problem

Given $n$ cities:

Choose a first city $n$
Choose a second $n-1$
Choose a third $n-2$
... $n!$
The Traveling Salesman Problem

Can we do better than \( n! \)

- First city doesn’t matter.
- Order doesn’t matter.

So we get \((n-1!)/2\).
The Growth Rate of $n!$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>11</td>
<td>479001600</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>12</td>
<td>6227020800</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>13</td>
<td>87178291200</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>14</td>
<td>1307674368000</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>720</td>
<td>15</td>
<td>20922789888000</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5040</td>
<td>16</td>
<td>355687428096000</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>40320</td>
<td>17</td>
<td>6402373705728000</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>362880</td>
<td>18</td>
<td>121645100408832000</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3628800</td>
<td>19</td>
<td>2432902008176640000</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>39916800</td>
<td>36</td>
<td>$3.6 \cdot 10^{41}$</td>
<td></td>
</tr>
</tbody>
</table>
Putting it into Perspective

<table>
<thead>
<tr>
<th></th>
<th>n^{22n}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed of light</td>
<td>3 \cdot 10^8 m/sec</td>
</tr>
<tr>
<td>Width of a proton</td>
<td>10^{-15} m</td>
</tr>
<tr>
<td>At one operation in the time it takes light to cross a proton</td>
<td>3 \cdot 10^{23} ops/sec</td>
</tr>
<tr>
<td>Since Big Bang</td>
<td>3 \cdot 10^{17} sec</td>
</tr>
<tr>
<td>Ops since Big Bang</td>
<td>9 \cdot 10^{40} ops</td>
</tr>
<tr>
<td>Neurons in brain</td>
<td>10^{11}</td>
</tr>
<tr>
<td>Parallel ops since Big Bang</td>
<td>9 \cdot 10^{51}</td>
</tr>
</tbody>
</table>

A dynamic programming solution: n^{22n}.
The Traveling Salesman Problem
Tackling Hard Problems

1. Use a technique that is guaranteed to find an optimal solution and likely to do so quickly. Linear programming:

The Concorde TSP Solver found an optimal route that visits 24,978 cities in Sweden.
2. Use a technique that is guaranteed to run quickly and find a “good” solution.

- The World Tour Problem

Does it make sense to insist on true optimality if the description of the original problem was approximate?
Modern TSP

From: http://xkcd.com/399/
The Complexity Zoo

The attempt to characterize the decidable languages by their complexity:

http://qwiki.caltech.edu/wiki/Complexity_Zoo
Characterizing Problems as Languages

- \((ab)^*\)
- \(WW^R = \{ww^R : w \in \{a, b\}^*\}\)
- \(WW = \{ww : w \in \{a, b\}^*\}\)
- \(SAT = \{w : w \text{ is a wff in Boolean logic and } w \text{ is satisfiable}\}\)

The theory only applies to decidable languages. So we will have nothing to say about:

- \(H = \{<M, w> : \text{Turing machine } M \text{ halts on input string } w\}\),
or
- \(PCP = \{<P> : \text{the Post Correspondence Problem instance } P \text{ has a solution}\}\).
All Problems Are Decision Problems

The Towers of Hanoi

Requires at least enough time to write the solution.

By restricting our attention to decision problems, the length of the answer is not a factor.
Encoding Types Other Than Strings

The length of the encoding matters.

**Integers:** use any base other than 1.

\[ 111111111111 \quad \text{vs} \quad 1100 \]
\[ 111111111111111111111111111111 \quad \text{vs} \quad 11110 \]

\[ \log_a x = \log_a b \log_b x \]

- \( \text{PRIMES} = \{ w : w \text{ is the binary encoding of a prime number} \} \)
Encoding Types Other Than Strings

**Graphs:** use an adjacency matrix:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Or a list of edges:

101/1/11/11/10/10/100/100/101
Graph Languages

- CONNECTED = \{<G> : G is an undirected graph and G is connected}\}.

- HAMILTONIANCIRCUIT = \{<G> : G is an undirected graph that contains a Hamiltonian circuit\}.
Characterizing Optimization Problems as Languages

- TSP-DECIDE = \{<G, cost> : <G> encodes an undirected graph with a positive distance attached to each of its edges and G contains a Hamiltonian circuit whose total cost is less than <cost>\}.
Choosing A Model of Computation

How many operations does this program execute?

\[ \text{tally} \ (A: \text{vector of } n \text{ integers}, \ n: \text{integer}) = \]
\[ \text{result} = 1 \]
For \( i = 1 \) to \( n \) do:
\[ \text{result} = \text{result} \times A[i]. \]
end
Return \( \text{result} \).
Choosing A Model of Computation

We’ll use Turing machines:

- Tape alphabet size?
- How many tapes?
- Deterministic vs. nondeterministic?
Measuring Time and Space Requirements

$timereq(M)$ is a function of $n$:

- If $M$ is a **deterministic** Turing machine that halts on all inputs, then:

  $$timereq(M) = f(n) = \text{the maximum number of steps that } M \text{ executes on any input of length } n.$$
Measuring Time and Space Requirements

- If $M$ is a **nondeterministic** Turing machine all of whose computational paths halt on all inputs, then:

  
  \[
  \begin{align*}
  & s, \underline{abab} \\
  & q_2, \#abab \\
  & q_1, \underline{abab} \\
  & q_1, \underline{abab} \\
  & q_3, \underline{bbab}
  \end{align*}
  \]

  
  $\text{timereq}(M) = f(n) = \text{the number of steps on the longest path that } M \text{ executes on any input of length } n.$
Measuring Time and Space Requirements

\( \text{spacereq}(M) \) is a function of \( n \):

- If \( M \) is a **deterministic** Turing machine that halts on all inputs, then:

  \[
  \text{spacereq}(M) = f(n) = \text{the maximum number of tape squares that } M \text{ reads on any input of length } n.
  \]

- If \( M \) is a **nondeterministic** Turing machine all of whose computational paths halt on all inputs, then:

  \[
  \text{spacereq}(M) = f(n) = \text{the maximum number of tape squares that } M \text{ reads on any path that it executes on any input of length } n.
  \]
$M$ Accepts $A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$

$L = \{a^n b^n c^n : n \geq 0\}$

Example: $\square aabbcc\square\square\square\square\square\square\square\square\square$  

Example: $\square aacccb\square\square\square\square\square\square\square\square\square$
$M$ Accepts $A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$

1. Move right onto $w$. If the first character is $\square$, halt and accept.

2. Loop:
   
2.1. Mark off an $a$ with a $1$.
2.2. Move right to the first $b$ and mark it off with a $2$. If there isn’t one or if there is a $c$ first, halt and reject.
2.3. Move right to the first $c$ and mark it off with a $3$. If there isn’t one or there is an $a$ first, halt and reject.
2.4. Move all the way back to the left, then right again past all the $1$’s (the marked off $a$’s). If there is another $a$, go back to the top of the loop. If there isn’t, exit the loop.

3. All $a$’s have found matching $b$’s and $c$’s and the read/write head is just to the right of the region of marked off $a$’s. Continue moving left to right to verify that all $b$’s and $c$’s have been marked. If they have, halt and accept. Otherwise halt and reject.
Analyzing the $A^nB^nC^n$ Machine - Time

If $w \in A^nB^nC^n$, the loop will be executed $n/3$ times.

Each time through the loop, the average number of steps executed is $2(n/3 + n/3 + n/6)$.

Then $M$ must make one final sweep all the way through $w$. That takes an additional $n$ steps.

So the total number of steps $M$ executes is:

$$2(n/3)(n/3 + n/3 + n/6) + n.$$ 

If $w \not\in A^nB^nC^n$, the number of steps executed by $M$ is lower. So:

$$\text{timereq}(M) = 2(n/3)(n/3 + n/3 + n/6) + n.$$ 

So the time required to run $M$ on an input of length $n$ grows as $n^2$. 
Analyzing the $A^nB^nC^n$ Machine - Space

$M$ uses only those tape squares that contain its input string, plus the blank on either side of it. So we have:

$$\text{spacereq}(M) =$$
Growth Rates of Functions
Asymptotic Dominance

- $f(n) \in \mathcal{O}(g(n))$
- $f(n) \in \Omega(g(n))$
- $f(n) \in \Theta(g(n))$
Asymptotic Dominance - $\mathcal{O}$

$f(n) \in \mathcal{O}(g(n))$ iff there exists a positive integer $k$ and a positive constant $c$ such that:

$$\forall n \geq k \ (f(n) \leq c \ g(n)).$$

Alternatively, if the limit exists:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

Or, $g$ grows at least as fast as $f$ does.
Asymptotic Dominance - $\mathcal{O}$

- $n^3 \in \mathcal{O}(n^3)$.
- $n^3 \in \mathcal{O}(n^4)$.
- $3n^3 \in \mathcal{O}(n^3)$.
- $n^3 \in \mathcal{O}(3^n)$.
- $n^3 \in \mathcal{O}(n!)$.
- $\log n \in \mathcal{O}(\text{ })$. 


Using $\mathcal{O}$

Suppose $\text{timereq}(M) = 3n^2 + 23n + 100$.

$\text{timereq}(M) \in \mathcal{O}(n^2)$:

Let $c = 4$ and $k = 28$:

\[
\forall n \geq 28 \quad (3n^2 + 23n + 100 \leq 4n^2) \\
(3(28^2)+23\cdot28+100 \leq 4\cdot28^2) \\
(2352 + 644 + 100 \leq 3136) \\
(3096 \leq 3136)
\]
Using $\Theta$ : Choosing $c$ and $k$

Suppose $\text{timereq}(M) = 3n^2 + 23n + 100$.

$\text{timereq}(M) \in \Theta(n^2)$:

Notice that:

\[3n^2 + 23n + 100 \leq 3n^2 + 23n^2 + 100n^2 = 126n^2.\]

So let $k = 1$ and $c = 126$. 
Using $\mathcal{O}$

Suppose $\text{timereq}(M) = 3n^2 + 23n + 100$.

$\text{timereq}(M) \in \mathcal{O}(n^3)$:

Let $c = 4$ and $k = 28$ again.

Or let $c = 3$ and $k = 5$. 
Using $\mathcal{O}$

1. $f(n) \in \mathcal{O}(f(n))$.

2. Addition:
   2.1. $\mathcal{O}(f(n)) = \mathcal{O}(f(n) + c_0)$ (if $1 \in \mathcal{O}(f(n))$).
   2.2. If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$, then $f_1(n) + f_2(n) \in \mathcal{O}(g_1(n) + g_2(n))$.
   2.3. $\mathcal{O}(f_1(n) + f_2(n)) = \mathcal{O}(\max(f_1(n), f_2(n)))$.

$$
2n + 3n \quad \in \mathcal{O}(\quad + \quad)
\in \mathcal{O}(\quad)

2n + 3n^2 \quad \in \mathcal{O}(\quad + \quad)
\in \mathcal{O}(\quad)
$$
Using $\mathcal{O}$

3. Multiplication:
   3.1. $\mathcal{O}(f(n)) = \mathcal{O}(c_0 \cdot f(n))$.
   3.2. If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$, then $f_1(n) \cdot f_2(n) \in \mathcal{O}(g_1(n) \cdot g_2(n))$.

$$2n \cdot 3n \in \mathcal{O}(\quad)$$
Using $\mathcal{O}$

4. Polynomials:
   4.1. If $a \leq b$ then $\mathcal{O}(n^a) \subseteq \mathcal{O}(n^b)$.
   4.2. If $f(n) = c_jn^j + c_{j-1}n^{j-1} + \ldots + c_1n + c_0$, then $f(n) \in \mathcal{O}(n^j)$.

   \[ 3n^2 + 23n + 100 \in \mathcal{O}(\quad) \]

5. Logarithms:
   5.1. For $a$ and $b > 1$, $\mathcal{O}(\log_a n) = \mathcal{O}(\log_b n)$.
   5.2. If $0 < a < b$ and $c > 1$, then $\mathcal{O}(n^a) \subseteq \mathcal{O}(n^a \log_c n) \subseteq \mathcal{O}(n^b)$.

   \[ 3n^2 \log_c n \in \mathcal{O}(\quad) \]
6. Exponentials (dominate polynomials):
   6.1. If $1 < a \leq b$ then $\mathcal{O}(a^n) \subseteq \mathcal{O}(b^n)$.
   6.2. If $a > 0$ and $b > 1$ then $\mathcal{O}(n^a) \subseteq \mathcal{O}(b^n)$.
   6.3. If $f(n) = c_{j+1}2^n + c_j n^j + c_{j-1} n^{j-1} + \ldots + c_1 n + c_0$,
       then $f(n) \in \mathcal{O}(2^n)$.
   6.4. $\mathcal{O}(n^r 2^n) \subseteq \mathcal{O}()$, for some $s$.

   \[
   2^n \in \mathcal{O}(\quad) \\
   3n^2 + 23n + 100 \in \mathcal{O}(\quad)
   \]

7. Factorial dominates exponentials: If $a \geq 1$,
   then $\mathcal{O}(a^n) \subseteq \mathcal{O}(n!)$.

   \[
   2^n \in \mathcal{O}(\quad)
   \]

8. Transitivity: If $f(n) \in \mathcal{O}(f_1(n))$ and $f_1(n) \in \mathcal{O}(f_2(n))$,
   then $f(n) \in \mathcal{O}(f_2(n))$. 

Using $\mathcal{O}$
Summarizing $\mathcal{O}$

$\mathcal{O}(c) \subseteq \mathcal{O}(\log_a n) \subseteq \mathcal{O}(n^b) \subseteq \mathcal{O}(d^n) \subseteq \mathcal{O}(n!)$
Asymptotic strong upper bound: \( f(n) \in \sigma(g(n)) \) iff, for every positive \( c \), there exists a positive integer \( k \) such that:

\[
\forall n \geq k \ (f(n) < c \ g(n)).
\]

Alternatively, if the limit exists:

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0
\]

In this case, we’ll say that \( f \) is “little-oh” of \( g \) or that \( g \) grows strictly faster than \( f \) does.
Asymptotic lower bound: \( f(n) \in \Omega(g(n)) \) iff there exists a positive integer \( k \) and a positive constant \( c \) such that:

\[
\forall n \geq k \ (f(n) \geq c \ g(n)).
\]

In other words, ignoring some number of small cases (all those of size less than \( k \)), and ignoring some constant factor \( c \), \( f(n) \) is bounded from below by \( g(n) \).

Alternatively, if the limit exists:

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0
\]

In this case, we’ll say that \( f \) is “big-Omega” of \( g \) or that \( g \) grows no faster than \( f \).
\( \omega \)

- **Asymptotic strong lower bound:** \( f(n) \in \omega(g(n)) \)
  
  iff, for every positive \( c \), there exists a positive integer \( k \) such that:

  \[ \forall n \geq k \ (f(n) > c \ g(n)). \]

Alternatively, if the required limit exists:

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty
\]

In this case, we’ll say that \( f \) is “little-omega” of \( g \) or that \( g \) grows strictly slower than \( f \) does.
\[
f(n) \in \Theta(g(n)) \text{ iff there exists a positive integer } k \text{ and positive constants } c_1, \text{ and } c_2 \text{ such that:}
\]
\[
\forall n \geq k \ (c_1 g(n) \leq f(n) \leq c_2 g(n))
\]
Or:
\[
f(n) \in \Theta(g(n)) \text{ iff:}
\]
\[
f(n) \in O(g(n)), \text{ and}
\]
\[
g(n) \in O(f(n)).
\]
Or:
\[
f(n) \in \Omega(g(n)).
\]

Is \(n^3 \in \Theta(n^3)\)?
Is \(n^3 \in \Theta(n^4)\)?
Is \(n^3 \in \Theta(n^5)\)?
Algorithmic Gaps

We’d like to show:

1. Upper bound: There exists an algorithm that decides $L$ and that has complexity $C_1$.
2. Lower bound: Any algorithm that decides $L$ must have complexity at least $C_2$.
3. $C_1 = C_2$.

If $C_1 = C_2$, we are done. Often, we’re not done.
Algorithmic Gaps

Example: TSP

- Upper bound: \( \text{timereq} \in \mathcal{O}(2^{n^k}) \).
- Don’t have a lower bound that says polynomial isn’t possible.

We group languages by what we know. And then we ask: “Is class \( CL_1 \) equal to class \( CL_2 \)?”
A Simple Example of Polynomial Speedup

Given a list of $n$ numbers, find the minimum and the maximum elements in the list. Or, as a language recognition problem:

$L = \{<\text{list of numbers}, \text{number}_1, \text{number}_2> : \text{number}_1 \text{ is the minimum element of the list and } \text{number}_2 \text{ is the maximum element}\}$.

$(23, 45, 73, 12, 45, 197; 12; 197) \in L$. 
A Simple Example of Polynomial Speedup

The straightforward approach:

\[
simplecompare(list: list \text{ of numbers}) =
\]
\[
max = list[1].
\]
\[
min = list[1].
\]
For \( i = 2 \) to \text{length}(list) do:
\[
\quad \text{If } list[i] < min \text{ then } min = list[i].
\]
\[
\quad \text{If } list[i] > max \text{ then } max = list[i].
\]

Requires \( 2(n-1) \) comparisons. So \( simplecompare \) is \( O(n) \).

But we can solve this problem in \( (3/2)(n-1) \) comparisons.

How?
A Simple Example of Polynomial Speedup

```python
efficientcompare(list: list of numbers) =
max = list[1].
min = list[1].
For i = 3 to length(list) by 2 do:
    If list[i] < list[i-1] then:
        If list[i] < min then min = list[i].
        If list[i-1] > max then max = list[i-1].
    Else:
        If list[i-1] < min then min = list[i-1].
        If list[i] > max then max = list[i].
If length(list) is even then check the last element.

Requires 3/2(n-1) comparisons.
```
String Search

\[ t: \quad a \ b \ c \ a \ b \ a \ b \ c \ a \ b \ d \]

\[ p: \quad a \ b \ c \ d \]

\[ a \ b \ c \ d \]

\[ a \ b \ c \ d \]

\[ a \ b \ c \ d \]

\[ \ldots \]
String Search

\[ \text{simple-string-search}(t, p: \text{strings}) = \]
\[ \begin{align*}
   & i = 0. \\
   & j = 0. \\
   & \text{While } i \leq |t| - |p| \text{ do:} \\
   & \quad \text{While } j < |p| \text{ do:} \\
   & \quad \quad \text{If } t[i+j] = p[j] \text{ then } j = j + 1. \\
   & \quad \quad \text{Else exit this loop.} \\
   & \quad \text{If } j = |p| \text{ then halt and accept.} \\
   & \quad \text{Else:} \\
   & \quad \quad i = i + 1. \\
   & \quad \quad j = 0. \\
   & \text{Halt and reject.}
\end{align*} \]

Let \( n \) be \(|t|\) and let \( m \) be \(|p|\). In the worst case (in which it doesn’t find an early match), \textit{simple-string-search} will go through its outer loop almost \( n \) times and, for each of those iterations, it will go through its inner loop \( m \) times.

So \( \text{timereq}(\text{simple-string-search}) \in \mathcal{O}(nm) \).
string-search-using-FSMs(t, p: strings) =
    1. Build the simple nondeterministic FSM $M$ that accepts any string that contains $p$ as a substring.
    2. Let $M' = ndfsmtodfsm(M)$.
    3. Let $M'' = minDFSM(M')$.
    4. Run $M''$ on $t$.
    5. If it accepts, accept. Else reject.

Step 4 runs in $n$ steps.

Steps 1-3 need only be done once for each pattern $p$. The resulting machine $M''$ can then be used to scan as many input strings as we want. But steps 1-3 are expensive since the number of states of $M'$ may grow exponentially with the number of states of $M$ (i.e., with the number of characters in $p$).
Matching time: $O(n)$.

Efficient preprocessing of pattern $p$.

$$
\begin{array}{ccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
t: & a & b & c & a & b & a & b & c & a & b & d \\
p: & a & b & c & a & b & d \\
\end{array}
$$

*But it already knows the first five characters of $t$.*

*Simple-string-search* will increment $i$ by 1 and then check:

$$
\begin{array}{ccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
t: & a & b & c & a & b & a & b & c & a & b & d \\
p: & a & b & c & a & b & d \\
\end{array}
$$
Exploiting the Kernel

When a match fails, $j$ is the number of characters that were successfully matched before the failure was detected. Ignore the first matched character. Call the remaining $j-1$ characters the \textit{kernel}.

\begin{itemize}
  \item $0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8$
  \item $t: \ a \ b \ c \ a \ b \ a \ b \ c \ a \ b \ d$
  \item $p: \ a \ b \ c \ a \ b \ d$
  \item $\times$
\end{itemize}

The kernel is $b \ c \ a \ b$.
Exploiting the Kernel

To find the next match, slide the beginning of the pattern to the right until it is just past the kernel and then back to the left to account for any overlap between the end of the kernel and the beginning of the pattern.

Initially:

\[
\begin{array}{cccc}
  b & c & a & b \\
  a & b & c & a & b & d \\
\end{array}
\]

After sliding left:

\[
\begin{array}{cccc}
  b & c & a & b \\
  a & b & c & a & b & d \\
\end{array}
\]

So the next match should place the pattern \( j \) (i.e., 5) characters to the right, minus the 2 overlap characters.
Exploiting the Kernel

The next match:

<table>
<thead>
<tr>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td>t:</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

$p:$

| a | b | c | a | b | d |

Note: this analysis of sliding distance is independent of the text string $t$. So we can preprocess a pattern $p$ and store, in a table $T$, the overlap numbers are for each value of $j$.

For the pattern $abcabd$, $T$ will be

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi[j]$</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

| kernel | $\varepsilon$ | $\varepsilon$ | $b$ | $bc$ | $bca$ | $bcab$ |
Exploiting the Kernel

The next match:

<table>
<thead>
<tr>
<th></th>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>t</strong>:</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td><strong>p</strong>:</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We don’t need to reset $j$ to 0. We already know that the first 2 characters of $p$ match. We use the old $j$ (i.e., 5) as an index into $T$ to see where to start. $\pi[j]$ in this case is 2.

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi[j]$</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>kernel</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>b</td>
<td>bc</td>
<td>bca</td>
<td>bcab</td>
</tr>
</tbody>
</table>

So we use $\pi[j]$ for two things:
- Computing the left sliding distance for the pattern.
- Deciding where to start checking at the next match.
Knuth-Morris-Pratt

* Knuth-Morris-Pratt*(t, p: strings) =

\[
i = 0.
\]
\[
j = 0.
\]
While \(i \leq |t| - |p|\) do:
    \[
    \text{While } j < |p| \text{ do:}
    \]
    If \(t[i+j] = p[j]\) then \(j = j + 1\).
    Else exit this loop.
    If \(j = |p|\) then halt and accept.
    Else:
    * \(i = i + j - T[j]\).
    * \(j = \max(0, T[j])\).

Halt and reject.

\(T[0]\) is -1 because, if a match fails immediately, the pattern must be shifted one character to the right for the next match. So we must treat \(j = 0\) as a special case in computing the next value for \(j\). That value must be 0, not -1. Thus the use of the \(\max\) function in the expression that defines the next value for \(j\).
An Example

Begin:

<table>
<thead>
<tr>
<th></th>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>t:</strong></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td><strong>p:</strong></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ i = 0, \; j = 0. \]

Compute new values for next match:
\[ i = i + j - T[j] = 0 + 5 - 2 = 3. \]
\[ j = \max(0, \; T[j]) = 2. \]

<table>
<thead>
<tr>
<th></th>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>t:</strong></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td><strong>p:</strong></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ i = 3, \; j = 2. \]

Compute new values for next match:
\[ i = i + j - T[j] = 3 + 2 - 0 = 5. \]
\[ j = \max(0, \; T[j]) = 0. \]

<table>
<thead>
<tr>
<th></th>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>t:</strong></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td><strong>p:</strong></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another Example

Let $p = \text{aaaaaab}$. We can only shift 1 on each match. $T$ is:

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T[j]$</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>kernel</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>a</td>
<td>aa</td>
<td>aaa</td>
<td>aaaa</td>
<td>aaaaa</td>
</tr>
</tbody>
</table>

$0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ ...$

$t$: a a a a a a a a a a a a a a a a a b  \quad i = 0, j = 0.$

$p$: a a a a a a b

$\times$

$i = 0, j = 6.$

Compute new values for next match:

$i = i + j - T[j] = 0 + 6 - 5 = 1  \quad j = max(0, T[j]) = 5$

$t$: a a a a a a a a a a a a a a a a a b

$p$: a a a a a a a b

$\times$

$i = 1, j = 6.$

But we don’t have to start $j$ over at 0. So we still beat simple-string-search.
Given: A text $t$ of length $n$, A pattern $p$ of length $m$, and The table $T$.

Consider each character $c$ of $t$. If the first comparison of $p$ to $c$ succeeds, then one of the following things must happen next:

- The rest of the pattern also matches. No further match attempts will be made so $c$ will never be examined again.
- Somewhere later the pattern fails and $c$ becomes part of the kernel. No kernel characters are ever reexamined. So $c$ will never be examined again.

So the number of successful comparisons is no more than $n$. 

Complexity of Knuth-Morris-Pratt
Complexity of Knuth-Morris-Pratt

The number of unsuccessful comparisons is also no more than $n$ since every unsuccessful comparison forces the process to stop and start over, sliding the pattern at least one character to the right. That can happen no more than $n$ times.

So the total number of comparisons is no more than $2n$ and so is $O(n)$. 
Building the Table $T$

We build up the entries in $T$ one at a time starting with $T[2]$ (since $T[0]$ is always -1 and $T[1]$ is always 0).

Assume that we have already considered a kernel of length $k-1$ and we are now considering one of length $k$. This new kernel is identical to the previous one except that one more character from $p$ has been added to the right.

Example:

Kernel:

\[
\begin{array}{c}
 b \\
 c \\
 a \\
 b \\
 c \\
 a \\
 b \\
 d \\
\end{array}
\]

Pattern:

\[
\begin{array}{c}
 a \\
 b \\
 c \\
 a \\
 b \\
 d \\
\end{array}
\]

To form the next longer kernel we add $a$ $b$ to the right of the previous kernel:

Kernel:

\[
\begin{array}{c}
 b \\
 c \\
 a \\
 b \\
 a \\
 b \\
 c \\
 a \\
 b \\
 d \\
\end{array}
\]

Pattern:

\[
\begin{array}{c}
 a \\
 b \\
 c \\
 a \\
 b \\
 d \\
\end{array}
\]
Building the Table $T$

There is no chance that there is now an overlap that starts to the left of the one we found at the last step. So there are only three possibilities:

- The match we found at the previous step can be extended by one character. The value of $T$ for the current kernel is one more than it was for the last one.
- The match we found on the previous step cannot be extended. In that case, we check to see whether a new, shorter match can be started.
- Neither can the old match be extended nor a new one started. In this case, the value of $T$ corresponding to the current kernel is 0.
Building the Table $T$

$buildoverlap(p: \text{pattern string}) =$

$T[0] = -1.$
$T[1] = 0.$

$j = 2$
$k = 0.$

While $j < |p|$ do:

- Compare $p[j - 1]$ to $p[k]$.
- If they are equal then:
  - $T[j] = k + 1.$
  - $j = j + 1.$
  - $k = k + 1.$

- If they are not equal but $k > 0$ then:
  - $k = T[k].$

- If they are not equal and $k = 0$ then:
  - $T[j] = 0.$
  - $j = j + 1.$
  - $k = 0.$

$buildoverlap$ executes at most $2m$ comparisons.
Replacing an Exponential Algorithm with a Polynomial One

● Context-free parsing can be done in $O(n^3)$ time instead of $O(2^n)$ time.

● Finding the highest probability path through a hidden Markov model can be done in $O(k^2n)$ time instead of $O(2^n)$ time.

● Finding the greatest common divisor of two integers can be done in $O(\log_2(\max(n, m)))$ time instead of exponential time.
GCD

gcd-obvious(n, m: integers) =
  1. Compute the prime factors of both n and m.
  2. Let k be the product of all factors common to n and m
      (including duplicates).
  3. Return k.

Example: The prime factors of 40 are \{2, 2, 2, 5\}.
The prime factors of 60 are \{2, 2, 3, 5\}.

So \(gcd(40, 60) = 2 \cdot 2 \cdot 5 = 20\).

But no efficient algorithm for prime factorization is known.
Euclid’s Algorithm

gcd-Euclid(n, m: integers) =
    If m = 0 return n.
    Else return gcd-Euclid(m, n (mod m)).

Example:

gcd-Euclid(2546, 1542) =
gcd-Euclid(1542, 984) =
gcd-Euclid(984, 558) =
gcd-Euclid(558, 426) =
gcd-Euclid(426, 132) =
gcd-Euclid(132, 30) =
gcd-Euclid(30, 12) =
gcd-Euclid(12, 6) =
gcd-Euclid(6, 0) =
6

Try it yourself.
Euclid’s Algorithm

\[ \text{gcd-Euclid}(n, m: \text{integers}) = \]
\[ \quad \text{If } m = 0 \text{ return } n. \]
\[ \quad \text{Else return } \text{gcd-Euclid}(m, n \mod m). \]

\[ \text{gcd-Euclid must halt:} \]

\[ n \mod m < m. \] So the second argument to \text{gcd-Euclid} is strictly decreasing. Since it can never become negative, it must eventually become 0.

\[ \text{gcd-Euclid halts with the correct result:} \]

For any integers \( n \) and \( m \), there exists some natural number \( j \) such that \( n = jm + (n \mod m) \). So, if some integer \( k \) divides both \( n \) and \( m \) it must also divide \( n \mod m \).
The Complexity of Euclid’s Algorithm

\( gcd-Euclid(n, m: \text{ integers}) = \)

If \( m = 0 \) return \( n \).
Else return \( gcd-Euclid(m, n \mod m) \).

Notice that \( n \mod m \leq n/2 \):

- \( m \leq n/2 \): \( n \mod m < m \leq n/2 \) and thus \( n \mod m \leq n/2 \).
- \( m > n/2 \): \( n \mod m = n - m \). So \( n \mod m \leq n/2 \).

\( gcd-Euclid \) swaps its arguments on each recursive call. So, after each pair of calls, the second argument is cut at least in half. Thus, after at most \( 2 \cdot \log_2 m \) calls, the second argument will be equal to 0 and \( gcd-Euclid \) will halt. If we assume that each division has constant cost, then:

\[ \text{timereq}(gcd-Euclid) \in \mathcal{O}(\log_2(\max(n, m))). \]
Depth-first search:

Space complexity:

Time complexity:

Problem:
Time-Space Tradeoffs - Search

**Breadth-first search:**

Space complexity:

Time complexity:
Time-Space Tradeoffs - Search

*Iterative-deepening search:*

Space complexity:

Time complexity: