1 Algorithm Analysis
2 Growth rate functions
3 The properties of growth rate functions:
4 Importance of the growth rate
5 An example
Algorithm Analysis

- It means: estimating the resources required.
Algorithm Analysis

- It means: estimating the resources required.
- The resources of algorithms: time and space.
Algorithm Analysis

- It means: estimating the resources required.
- The resources of algorithms: time and space.
- We mainly consider time: harder to estimate; often more critical.

The efficiency of an algorithm is measured by a runtime function $T(n)$. $n$ is the size of the input. Strictly speaking, $n$ is the # of bits needed to represent input. Commonly, $n$ is the # of items in the input, if each item is of fixed size. This makes no difference in asymptotic analysis in most cases.
Algorithm Analysis

- It means: **estimating** the resources required.
- The **resources** of algorithms: **time** and **space**.
- We mainly consider **time**: harder to estimate; often more critical.
- The **efficiency** of an algorithm is measured by a runtime function $T(n)$. 

$n$ is the size of the input. Strictly speaking, $n$ is the number of bits needed to represent input. Commonly, $n$ is the number of items in the input, if each item is of fixed size. This makes no difference in asymptotic analysis in most cases.
Algorithm Analysis

- It means: estimating the resources required.
- The resources of algorithms: time and space.
- We mainly consider time: harder to estimate; often more critical.
- The efficiency of an algorithm is measured by a runtime function $T(n)$.
- $n$ is the size of the input.
Algorithm Analysis

- It means: estimating the resources required.
- The resources of algorithms: time and space.
- We mainly consider time: harder to estimate; often more critical.
- The efficiency of an algorithm is measured by a runtime function $T(n)$.
- $n$ is the size of the input.
- Strictly speaking, $n$ is the # of bits needed to represent input.
Algorithm Analysis

- It means: estimating the resources required.
- The resources of algorithms: time and space.
- We mainly consider time: harder to estimate; often more critical.
- The efficiency of an algorithm is measured by a runtime function $T(n)$.
- $n$ is the size of the input.
- Strictly speaking, $n$ is the # of bits needed to represent input.
- Commonly, $n$ is the # of items in the input, if each item is of fixed size.
It means: estimating the resources required.

The resources of algorithms: time and space.

We mainly consider time: harder to estimate; often more critical.

The efficiency of an algorithm is measured by a runtime function $T(n)$.

$n$ is the size of the input.

Strictly speaking, $n$ is the # of bits needed to represent input.

Commonly, $n$ is the # of items in the input, if each item is of fixed size.

This makes no difference in asymptotic analysis in most cases.
Example 1

An array of $k$ int. Strictly speaking $n = 32k$ bits. However, since int has fixed size of 32 bits, we can use $n = k$ as input size.
Example 1

An array of $k$ int. Strictly speaking $n = 32k$ bits. However, since int has fixed size of 32 bits, we can use $n = k$ as input size.

Example 2

The input is one integer of $k$ digits long. Since its size is not fixed ($k$ can be arbitrarily large). The input size is **not** $n = 1$. It is $n = \lceil \log(10^k) \rceil = 4k$ bits long.
What’s $T(n)$?

- Defining $T(n)$ as the real run time is meaningless, because the real run time depends on many factors, such as the machine speed, the programming language used, the quality of compilers etc. These are not the properties of the algorithm.

- $T(n) = \text{the number of basic instructions performed by the algorithm.}$

- Basic instructions: $\text{+,-,*,/}, \text{read from/write into a memory location, comparison, branching to another instruction ...}$

- These are not basic instructions: input/output statement, $\text{sin}(x), \text{exp}(x) ...$. These actions are done by function calls, not by a single machine instruction.

- Knowing $T(n)$ and the machine speed, we can estimate the real runtime.

Example 3: The machine speed is $10^8 \text{ins/sec}$. $T(n) = 10^6$. The real runtime would be about $10^{-8} \times 10^6 = 0.01 \text{sec.}$
What’s $T(n)$?

- Defining $T(n)$ as the real run time is meaningless, because the real run time depends on many factors, such as the machine speed, the programming language used, the quality of compilers etc. These are not the properties of the algorithm.
- $T(n) \overset{\text{def}}{=} \text{the number of basic instructions performed by the algorithm.}$
What's $T(n)$?

- Defining $T(n)$ as the real run time is meaningless, because the real run time depends on many factors, such as the machine speed, the programming language used, the quality of compilers etc. These are not the properties of the algorithm.

- $T(n) \overset{\text{def}}{=} \text{the number of basic instructions} \text{ performed by the algorithm.}$

- **Basic instructions:** $+, -, *, /$, read from/write into a memory location, comparison, branching to another instruction ...
What’s $T(n)$?

- Defining $T(n)$ as the real run time is meaningless, because the real run time depends on many factors, such as the machine speed, the programming language used, the quality of compilers etc. These are not the properties of the algorithm.

- $T(n) \overset{\text{def}}{=} \text{the number of basic instructions performed by the algorithm.}$

- **Basic instructions:** +, −, *, /, read from/write into a memory location, comparison, branching to another instruction ...

- These are not basic instructions: input/output statement, $\sin(x)$, $\exp(x)$ . . . These actions are done by function calls, not by a single machine instruction.

Knowing $T(n)$ and the machine speed, we can estimate the real runtime.

Example 3: The machine speed is $10^8 \text{ins/sec}$. $T(n) = 10^6$. The real runtime would be about $10^{-8} \times 10^6 = 0.01 \text{sec.}$
What’s $T(n)$?

- Defining $T(n)$ as the real run time is meaningless, because the real run time depends on many factors, such as the machine speed, the programming language used, the quality of compilers etc. These are not the properties of the algorithm.

- $T(n) \overset{\text{def}}{=} \text{the number of basic instructions performed by the algorithm.}$

- **Basic instructions:** $+, -, *, /$, read from/write into a memory location, comparison, branching to another instruction ...

- These are not basic instructions: input/output statement, $\sin(x), \exp(x)$ ... These actions are done by function calls, not by a single machine instruction.

- Knowing $T(n)$ and the machine speed, we can estimate the real runtime.

- Example 3: The machine speed is $10^8$ ins/sec. $T(n) = 10^6$. The real runtime would be about $10^{-8} \times 10^6 = 0.01$ sec.
Example 4: Consider this simple program:

1: \( s = 0 \)
2: \textbf{for} \( i = 1 \) \textbf{to} \( n \) \textbf{do} \\
3: \hspace{1em} \textbf{for} \( j = 1 \) \textbf{to} \( n \) \textbf{do} \\
4: \hspace{2em} \( s = s + i + j \) \\
5: \hspace{1em} \textbf{end for} \\
6: \textbf{end for} \\

It's hard to get the exact expression of \( T(n) \) even for this very simple program. Also, the exact value of \( T(n) \) depends on factors such as programming language, compiler. These are not the properties of the loop. They should not be our concern.

We can see: the loop iterates \( n^2 \) times, and loop body takes a constant number of instructions. So \( T(n) = an^2 + bn + c \) for some constants \( a, b, c \).

We say the growth rate of \( T(n) \) is \( n^2 \). This is the sole property of the algorithm and is our main concern.
Example 4: Consider this simple program:

1: \( s = 0 \)
2: for \( i = 1 \) to \( n \) do
3: \hspace{1em} for \( j = 1 \) to \( n \) do
4: \hspace{2em} \( s = s + i + j \)
5: \hspace{1em} end for
6: end for

- \( T(n) = \) ? It’s hard to get the exact expression of \( T(n) \) even for this very simple program.
Example 4: Consider this simple program:

1: \( s = 0 \)
2: \( \text{for } i = 1 \text{ to } n \text{ do} \)
3: \( \quad \text{for } j = 1 \text{ to } n \text{ do} \)
4: \( \quad \quad s = s + i + j \)
5: \( \quad \text{end for} \)
6: \( \text{end for} \)

- \( T(n) =? \) It’s hard to get the exact expression of \( T(n) \) even for this very simple program.
- Also, the exact value of \( T(n) \) depends on factors such as program language, compiler. These are not the properties of the loop. They should not be our concern.
Example 4: Consider this simple program:

1: \( s = 0 \)
2: for \( i = 1 \) to \( n \) do
3: for \( j = 1 \) to \( n \) do
4: \( s = s + i + j \)
5: end for
6: end for

- \( T(n) = ? \) It’s hard to get the exact expression of \( T(n) \) even for this very simple program.
- Also, the exact value of \( T(n) \) depends on factors such as prog language, compiler. These are not the properties of the loop. They should not be our concern.
- We can see: the loop iterates \( n^2 \) times, and loop body takes constant number of instructions.
Example 4: Consider this simple program:

1: \( s = 0 \)
2: \( \text{for } i = 1 \text{ to } n \text{ do } \)
3: \( \quad \text{for } j = 1 \text{ to } n \text{ do } \)
4: \( \quad \quad s = s + i + j \)
5: \( \quad \text{end for} \)
6: \( \text{end for} \)

- \( T(n) = ? \) It’s hard to get the exact expression of \( T(n) \) even for this very simple program.
- Also, the exact value of \( T(n) \) depends on factors such as prog language, compiler. These are not the properties of the loop. They should not be our concern.
- We can see: the loop iterates \( n^2 \) times, and loop body takes constant number of instructions.
- So \( T(n) = an^2 + bn + c \) for some constants \( a, b, c \).
Example 4: Consider this simple program:

1: \( s = 0 \)

2: for \( i = 1 \) to \( n \) do

3: \quad for \( j = 1 \) to \( n \) do

4: \quad \quad \( s = s + i + j \)

5: \quad end for

6: end for

- \( T(n) = \) ? It’s hard to get the exact expression of \( T(n) \) even for this very simple program.

- Also, the exact value of \( T(n) \) depends on factors such as prog language, compiler. These are not the properties of the loop. They should not be our concern.

- We can see: the loop iterates \( n^2 \) times, and loop body takes constant number of instructions.

- So \( T(n) = an^2 + bn + c \) for some constants \( a, b, c \).

- We say the growth rate of \( T(n) \) is \( n^2 \). This is the sole property of the algorithm and is our main concern.
Outline

1. Algorithm Analysis
2. Growth rate functions
3. The properties of growth rate functions:
4. Importance of the growth rate
5. An example
Growth rate functions

We want to define the precise meaning of growth rate.

Definition 1:

\[ \Theta(g(n)) = \{ f(n) \mid \exists c_1 > 0, c_2 > 0, n_0 \geq 0 \text{ so that } \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \} \]

If \( f(n) \in \Theta(g(n)) \), we also write \( f(n) = \Theta(g(n)) \) and say: the growth rate of \( f(n) \) is the same as the growth rate of \( g(n) \).
Example 5

\[ f(n) = \frac{1}{12} n^2 + 60n - 4 \in \Theta(n^2) \text{ (or write } f(n) = \Theta(n^2) \text{.)} \]

Proof: We need to find \( c_1 \) and \( n_0 \) so that \( \forall n \geq n_0, \)

\[ c_1 n^2 \leq \frac{1}{12} n^2 + 60n - 4 \]

Pick \( c_1 = 1/12 \), the above becomes: \( 0 \leq 60n - 4 \). This is true for all \( n \geq n_0 = 1 \). We also need to find \( c_2 \) and \( n_0 \) so that \( \forall n \geq n_0, \)

\[ \frac{1}{12} n^2 + 60n - 4 \leq c_2 n^2 \]

For any \( n \geq 1 \), we have:

\[ \frac{1}{12} n^2 + 60n - 4 < n^2 + 60n \leq n^2 + 60n^2 = 61n^2 \]

So if \( c_1 = 1/12, c_2 = 61 \) and \( n_0 = 1 \), all the required conditions hold.
Definition 2:

\[ O(g(n)) = \{ f(n) \mid \exists c_2 > 0, n_0 \geq 0 \text{ so that } \forall n \geq n_0, 0 \leq f(n) \leq c_2 g(n) \} \]

If \( f(n) \in O(g(n)) \), we also write \( f(n) = O(g(n)) \) and say: the growth rate of \( f(n) \) is at most the growth rate of \( g(n) \).

Example 6

\( f(n) = 10n - 4 \in O(0.01n^2) \) (or write \( f(n) = O(0.01n^2) \).)
Definition 3:

\[ \Omega(g(n)) = \{ f(n) \mid \exists c_1 > 0, n_0 \geq 0 \text{ so that } \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \} \]

If \( f(n) \in \Omega(g(n)) \), we also write \( f(n) = \Omega(g(n)) \) and say: the growth rate of \( f(n) \) is at least the growth rate of \( g(n) \).
Definition 4:

\[ o(g(n)) = \{ f(n) \mid \forall c > 0, \exists n_0 \geq 0 \text{ so that } \forall n \geq n_0, 0 \leq f(n) \leq cg(n) \} \]

If \( f(n) \in o(g(n)) \), we also write \( f(n) = o(g(n)) \) and say: the growth rate of \( f(n) \) is strictly less than the growth rate of \( g(n) \).

Example:

\( f(n) = 2n \) and \( g(n) = n^2 \). Then:
\( f(n) = O(g(n)), f(n) = o(g(n)), \) but \( f(n) \neq \Theta(g(n)) \),
Definition 5:

\[ \omega(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 \geq 0 \text{ so that } \forall n \geq n_0, 0 \leq cg(n) \leq f(n) \} \]

If \( f(n) \in \omega(g(n)) \), we also write \( f(n) = \omega(g(n)) \) and say: the growth rate of \( f(n) \) is strictly bigger than the growth rate of \( g(n) \).
Outline

1. Algorithm Analysis
2. Growth rate functions
3. The properties of growth rate functions:
4. Importance of the growth rate
5. An example
The properties of growth rate functions:

The meaning of these notations (roughly speaking):

<table>
<thead>
<tr>
<th>if</th>
<th>the growth-rate is</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(n) = \Theta(g(n))$</td>
<td>$=$</td>
</tr>
<tr>
<td>$f(n) = O(g(n))$</td>
<td>$\leq$</td>
</tr>
<tr>
<td>$f(n) = \Omega(g(n))$</td>
<td>$\geq$</td>
</tr>
<tr>
<td>$f(n) = o(g(n))$</td>
<td>$&lt;$</td>
</tr>
<tr>
<td>$f(n) = \omega(g(n))$</td>
<td>$&gt;$</td>
</tr>
</tbody>
</table>
Some properties of growth rate functions:

\[ f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \]

\[ f(n) = O(g(n)) \iff g(n) = \Omega(f(n)) \]

\[ f(n) = o(g(n)) \iff g(n) = \omega(f(n)) \]

\[ f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \implies f(n) = O(h(n)) \]

If we replace \( O \) by \( \Theta \), \( \Omega \), \( o \), \( \omega \), it holds true.
Some properties of growth rate functions:

\[ f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \]
Some properties of growth rate functions:

- $f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$
- $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
Some properties of growth rate functions:

- \( f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \)
- \( f(n) = O(g(n)) \iff g(n) = \Omega(f(n)) \)
- \( f(n) = o(g(n)) \iff g(n) = \omega(f(n)) \)
Some properties of growth rate functions:

- \( f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \)
- \( f(n) = O(g(n)) \iff g(n) = \Omega(f(n)) \)
- \( f(n) = o(g(n)) \iff g(n) = \omega(f(n)) \)
- \( f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \implies f(n) = O(h(n)) \)
Some properties of growth rate functions:

- \( f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \)
- \( f(n) = O(g(n)) \iff g(n) = \Omega(f(n)) \)
- \( f(n) = o(g(n)) \iff g(n) = \omega(f(n)) \)
- \( f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \implies f(n) = O(h(n)) \) if we replace \( O \) by \( \Theta, \Omega, o, \omega \), it holds true.
1. Algorithm Analysis

2. Growth rate functions

3. The properties of growth rate functions:

4. Importance of the growth rate

5. An example
The growth rate of the runtime function is the most important property of an algorithm. Assuming $10^9$ instruction/sec, the real runtime:

<table>
<thead>
<tr>
<th>$f(n)$</th>
<th>$n = 10$</th>
<th>30</th>
<th>50</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log_2 n$</td>
<td>3.3 ns</td>
<td>4.9 ns</td>
<td>5.6 ns</td>
<td>9.9 ns</td>
</tr>
<tr>
<td>$n$</td>
<td>10 ns</td>
<td>30 ns</td>
<td>50 ns</td>
<td>1 $\mu s$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>0.1 $\mu s$</td>
<td>0.9 $\mu s$</td>
<td>2.5 $\mu s$</td>
<td>1 ms</td>
</tr>
<tr>
<td>$n^3$</td>
<td>1 $\mu s$</td>
<td>27 $\mu s$</td>
<td>125 $\mu s$</td>
<td>1 sec</td>
</tr>
<tr>
<td>$n^5$</td>
<td>0.1 ms</td>
<td>24.3 ms</td>
<td>0.3 sec</td>
<td>277 h</td>
</tr>
<tr>
<td>$2^n$</td>
<td>1 $\mu s$</td>
<td>1 sec</td>
<td>312 h</td>
<td>$3.4 \cdot 10^{281}$ Cent</td>
</tr>
</tbody>
</table>

- If $T(n) = n^k$ for some constant $k > 0$, the runtime is polynomial.
- If $T(n) = a^n$ for some constant $a > 1$, the runtime is exponential.
• $T(n) = 2^n$, $n = 360$ and assuming $10^9$ instructions/sec.
\( T(n) = 2^n, \ n = 360 \) and assuming \( 10^9 \) instructions/sec.

\[ T(360) = 2^{360} = (2^{10})^{36} \approx (10^3)^{36} = 10^{108} \text{ instructions.} \]
- $T(n) = 2^n$, $n = 360$ and assuming $10^9$ instructions/sec.
- $T(360) = 2^{360} = (2^{10})^{36} \approx (10^3)^{36} = 10^{108}$ instructions.
- This translates into: $10^{99}$ CPU sec, about $3 \cdot 10^{91}$ years.

For comparison: the age of the universe: about $1.5 \cdot 10^{10}$ years.

The number of atoms in the known universe: $\leq 10^{80}$.

If every atom in the known universe is a supercomputer and starts at the beginning of the big bang, we have only done $1.5 \cdot 10^{10} \times 10^{80} = 5 \%$ of the needed computations!

Moore’s law: CPU speed doubles every 18 months. Then, instead of solving the problem of size $n$ = say 100, we can solve the problem of size 101.

An exponential time algorithm cannot be used to solve problems of realistic input size, no matter how powerful the computers are!
\( T(n) = 2^n, \ n = 360 \) and assuming \( 10^9 \) instructions/sec.

\( T(360) = 2^{360} = (2^{10})^{36} \approx (10^3)^{36} = 10^{108} \) instructions.

This translates into: \( 10^{99} \) CPU sec, about \( 3 \cdot 10^{91} \) years.

For comparison: the age of the universe: about \( 1.5 \cdot 10^{10} \) years.
\[ T(n) = 2^n, \quad n = 360 \text{ and assuming } 10^9 \text{ instructions/sec}. \]

\[ T(360) = 2^{360} = (2^{10})^{36} \approx (10^3)^{36} = 10^{108} \text{ instructions}. \]

This translates into: \(10^{99}\) CPU sec, about \(3 \cdot 10^{91}\) years.

For comparison: the age of the universe: about \(1.5 \cdot 10^{10}\) years.

The number of atoms in the known universe: \(\leq 10^{80}\).
• $T(n) = 2^n$, $n = 360$ and assuming $10^9$ instructions/sec.
• $T(360) = 2^{360} = (2^{10})^{36} \approx (10^3)^{36} = 10^{108}$ instructions.
• This translates into: $10^{99}$ CPU sec, about $3 \cdot 10^{91}$ years.
• For comparison: the age of the universe: about $1.5 \cdot 10^{10}$ years.
• The number of atoms in the known universe: $\leq 10^{80}$.
• If every atom in the known universe is a supercomputer and starts at the beginning of the big bang, we have only done
  \[
  \frac{1.5 \cdot 10^{10} \times 10^{80}}{3 \cdot 10^{91}} = 5\% \text{ of the needed computations!}
  \]
\[ T(n) = 2^n, \quad n = 360 \] and assuming \( 10^9 \) instructions/sec.

\[ T(360) = 2^{360} = (2^{10})^{36} \approx (10^3)^{36} = 10^{108} \text{ instructions.} \]

This translates into: \( 10^{99} \) CPU sec, about \( 3 \cdot 10^{91} \) years.

For comparison: the age of the universe: about \( 1.5 \cdot 10^{10} \) years.

The number of atoms in the known universe: \( \leq 10^{80} \).

If every atom in the known universe is a supercomputer and starts at the beginning of the big bang, we have only done

\[ \frac{1.5 \cdot 10^{10} \times 10^{80}}{3 \cdot 10^{91}} = 5\% \text{ of the needed computations!} \]

Moore’s law: CPU speed doubles every 18 months. Then, instead of solving the problem of size \( n = \text{say 100} \), we can solve the problem of size 101.
\[ T(n) = 2^n, \quad n = 360 \text{ and assuming } 10^9 \text{ instructions/sec.} \]

\[ T(360) = 2^{360} = (2^{10})^{36} \approx (10^3)^{36} = 10^{108} \text{ instructions.} \]

This translates into: \(10^{99}\) CPU sec, about \(3 \cdot 10^{91}\) years.

For comparison: the age of the universe: about \(1.5 \cdot 10^{10}\) years.

The number of atoms in the known universe: \(\leq 10^{80}\).

If every atom in the known universe is a supercomputer and starts at the beginning of the big bang, we have only done
\[
\frac{1.5 \cdot 10^{10} \times 10^{80}}{3 \cdot 10^{91}} = 5\% \text{ of the needed computations!}
\]

Moore’s law: CPU speed doubles every 18 months. Then, instead of solving the problem of size \(n = \text{say } 100\), we can solve the problem of size 101.

An exponential time algorithm cannot be used to solve problems of realistic input size, no matter how powerful the computers are!
Outline

1. Algorithm Analysis
2. Growth rate functions
3. The properties of growth rate functions:
4. Importance of the growth rate
5. An example
An example

Some simple looking problems indeed require exp runtime. Here is a very important application that depends on this fact.

P1: Factoring Problem
Input: an integer $X$.
Output: Find its prime factorization.
If $X = 117$, the output: $X = 3 \cdot 3 \cdot 13$.

P2: Primality Testing
Input: an integer $X$.
Output: "yes" if $X$ is a prime number; "no" if not.
If $X = 117$, output "no". If $X = 456731$, output = ?
Some **simple looking** problems indeed require exp runtime. Here is a very important application that depends on this fact.

<table>
<thead>
<tr>
<th>P1: Factoring Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: an integer $X$.</td>
</tr>
<tr>
<td>Output: Find its prime factorization.</td>
</tr>
</tbody>
</table>
An example

Some simple looking problems indeed require exp runtime. Here is a very important application that depends on this fact.

**P1: Factoring Problem**

Input: an integer $X$.
Output: Find its prime factorization.

If $X = 117$, the output: $X = 3 \cdot 3 \cdot 13$. 
Some simple looking problems indeed require exp runtime. Here is a very important application that depends on this fact.

**P1: Factoring Problem**

Input: an integer $X$.
Output: Find its prime factorization.

If $X = 117$, the output: $X = 3 \cdot 3 \cdot 13$.

**P2: Primality Testing**

Input: an integer $X$.
Output: "yes" if $X$ is a prime number; "no" if not.
Some simple looking problems indeed require exp runtime. Here is a very important application that depends on this fact.

**P1: Factoring Problem**

Input: an integer $X$.
Output: Find its prime factorization.

If $X = 117$, the output: $X = 3 \cdot 3 \cdot 13$.

**P2: Primality Testing**

Input: an integer $X$.
Output: "yes" if $X$ is a prime number; "no" if not.

- If $X = 117$, output "no".
Some simple looking problems indeed require exp runtime. Here is a very important application that depends on this fact.

### P1: Factoring Problem

<table>
<thead>
<tr>
<th>Input: an integer $X$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Find its prime factorization.</td>
</tr>
</tbody>
</table>

If $X = 117$, the output: $X = 3 \cdot 3 \cdot 13$.

### P2: Primality Testing

<table>
<thead>
<tr>
<th>Input: an integer $X$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: ”yes” if $X$ is a prime number; ”no” if not.</td>
</tr>
</tbody>
</table>

- If $X = 117$, output ”no”.
- If $X = 456731$, output = ?
P1 and P2 are related.

If we can solve P1, we can solve P2 immediately.
- P1 and P2 are related.
- If we can solve P1, we can solve P2 immediately.
- The reverse is not true: even if we know $X$ is not a prime, how to find its prime factors?

```c
Find-Factor(X)
1: if X is even then
2: return "2 is a factor"
3: end if
4: for i = 3 to $\sqrt{X}$ by +2 do
5: test if $X \% i = 0$, if yes, output "i is a factor"
6: end for
7: return "X is a prime."
```
P1 and P2 are related.
If we can solve P1, we can solve P2 immediately.
The reverse is not true: even if we know $X$ is not a prime, how to find its prime factors?
P1 is harder than P2.
• P1 and P2 are related.
• If we can solve P1, we can solve P2 immediately.
• The reverse is not true: even if we know $X$ is not a prime, how to find its prime factors?
• P1 is harder than P2.
• How to solve P1?

Find-Factor($X$):
1: if $X$ is even then
2: return "2 is a factor"
3: end if
4: for $i = 3$ to $\sqrt{X}$ by $+2$ do
5: test if $X \% i = 0$, if yes, output "i is a factor"
6: end for
7: return "$X$ is a prime."
P1 and P2 are related.

If we can solve P1, we can solve P2 immediately.

The reverse is not true: even if we know $X$ is not a prime, how to find its prime factors?

P1 is harder than P2.

How to solve P1?

**Find-Factor**($X$)

1: if $X$ is even then
2: return ”2 is a factor”
3: end if
4: for $i = 3$ to $\sqrt{X}$ by $+2$ do
5: test if $X \% i = 0$, if yes, output ”$i$ is a factor”
6: end for
7: return ”$X$ is a prime.”
To solve P1, we call \textbf{Find-Factor}(X) to find the smallest prime factor \( i \) of \( X \). Then call \textbf{Find-Factor}(X/i) ...
To solve P1, we call **Find-Factor**($X$) to find the smallest prime factor $i$ of $X$. Then call **Find-Factor**($X/i$) ...

The runtime of **Find-Factor**: $X$ is not a fixed-size object. So the input size $n$ is the \# of bits needed to represent $X$. 
To solve P1, we call **Find-Factor**\((X)\) to find the smallest prime factor \(i\) of \(X\). Then call **Find-Factor**\((X/i)\) ...

The runtime of **Find-Factor**: \(X\) is not a fixed-size object. So the input size \(n\) is the \# of bits needed to represent \(X\).

\(X\) is \(n\) bits long, the value of \(X\) is \(\leq 2^n\).
To solve P1, we call \textbf{Find-Factor}(X) to find the smallest prime factor \( i \) of \( X \). Then call \textbf{Find-Factor}(X/i) ...

The runtime of \textbf{Find-Factor}: \( X \) is not a fixed-size object. So the input size \( n \) is the \# of bits needed to represent \( X \).

\( X \) is \( n \) bits long, the value of \( X \) is \( \leq 2^n \).

In the worst case, we need to perform \( \frac{1}{2} \sqrt{2^n} = \frac{1}{2} (1.414)^n \) divisions. So \textbf{this is an exp time algorithm.}
To solve P1, we call \textbf{Find-Factor}($X$) to find the smallest prime factor $i$ of $X$. Then call \textbf{Find-Factor}($X/i$) ...

The runtime of \textbf{Find-Factor}: $X$ is not a fixed-size object. So the input size $n$ is the \# of bits needed to represent $X$.

$X$ is $n$ bits long, the value of $X$ is $\leq 2^n$.

In the worst case, we need to perform $\frac{1}{2}\sqrt{2^n} = \frac{1}{2}(1.414)^n$ divisions. So this is an exp time algorithm.

Minor improvements can be (and had been) made. But basically, we have to perform most of these tests. No poly-time algorithm for Factoring is known.

It is strongly believed, (but not proven), no poly-time algorithm for solving the Factoring problem exists.