Algorithms

Adnan YAZICI
Dept. of Computer Engineering
Middle East Technical Univ.
Ankara - TURKEY
INTRODUCTION

**Defn:** An *algorithm* is a complete set of rules that transform the input into the output in a finite number of steps. A rule typically calls for the execution of one or more operations. That is, an algorithm describes a specific computational procedure for solving a problem in a finite number of steps.

**Purpose of studying algorithm:**
- Choose the best alg. (or appropriate alg.)
- Learn techniques to develop new algorithms.

**Which algorithm** is best depends on:
- The size of the problem
- The type of the input – the extent to which the items are already somewhat sorted
- The kind of storage device to be used: main memory disks, or tapes
Aspects of studying algorithms:

1. Designing algorithms:
   - putting the pieces of the puzzles together,
   - choosing data structures,
   - selecting the basic approaches to the solution of the problem,

   The most popular design strategies are divide&conquer, greedy, dynamic prog., backtracking, and branch&bound.

2. Expressing and implementing the algorithm

Concerns are:
   - clearness
   - conciseness
   - effectiveness
3. Analyzing the algorithm

Algorithm analysis is assessing the time and space resources required by an algorithm as a function of the size of the problem, without actually implementing the algorithm.

4. Compare UB and LB to see if your solution is good enough

Analyzing the algorithm gives us the upper bound to solve the problem
Analyzing the problem gives us the lower bound to solve the problem

5. Validate the algorithm

We show that the algorithm computes the correct answer for all possible legal (or given) inputs
6. **Verifying the algorithm (or program)**
   An algorithm is said to be *correct* (verified) if, for every input instance, it halts with the correct output.

7. **Testing algorithms**
   - There are two phases;
     - **Debugging**: The process of executing programs on sample data sets to determine if faulty results occur, and if so, to correct them.
     - “Debugging can only point to the presence of errors, but not to their absence”

   - **Profiling**: the process of executing a correct program on various data sets and measuring the time (and space) it takes to compute the results.
A *running time function*, $T(n)$, yields the time required to execute the algorithm of a problem of size ‘n.’ Often the analysis of an algorithm leads to $T(n)$, which may contain unknown constants which depend on the characteristics of the ideal machine. So, we cannot determine this function exactly.

*Example:* $T(n) = an^2 + bn + c$, where $a, b, and c$ are unspecified constants

*Reasons why an absolute measure for an algorithm is not appropriate:*

- Execution time is sensitive to the amount of data manipulated, grows as the amount of data increases
- Because of the execution speed of the processors, the execution times for an algorithm on the same data set on two different computers may differ.
- Depending on how an algorithm is implemented on a particular computer (choice of PL, compiler, etc.) an implementation of an algorithm may run faster than another, even on the same computer with the same data set.
GROWTH RATE

The growth rate of $T(n)$ is referred to the computational complexity of the algorithm. The computational complexity gives a concise way of saying how the running time, $T(n)$, varies with $n$ and is independent of any particular implementation. The lower the growth rate, the faster the algorithm, at least for large values of $n$.

For example;

For $T(n) = an^2 + bn + c$, the growth rate is $O(n^2)$

The goal of the algorithm designer should be an algorithm with as low a growth rate of the running time function, $T(n)$, of that algorithm as possible.
Growth Rate (Cont.)

The typical growth rates for real life algorithms are:

- \( 1, \log n, n, n \log n, n^2, n^3, 2^n, n!, n^n \)

Problems with growth rate analysis:

An algorithm with a smaller growth rate will not run faster than one with a higher growth rate for any particular \( n \), but only for \( n \) ‘large enough’

Algorithms with identical growth rates may have strikingly different running times because of the constants in the running time functions.

The value of \( n \) where two growth rates are the same is called the break-even point.

We measure the complexity of an algorithm by identifying a basic operation and then counting how many times the algorithm performs that basic operation for an input size \( n \).

<table>
<thead>
<tr>
<th>problem</th>
<th>input of size ( n )</th>
<th>basic operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>searching a list</td>
<td>lists with ( n ) elements</td>
<td>comparison</td>
</tr>
<tr>
<td>sorting a list</td>
<td>lists with ( n ) elements</td>
<td>comparison</td>
</tr>
<tr>
<td>multiplying two matrices</td>
<td>two ( n )-by-( n ) matrices</td>
<td>multiplication</td>
</tr>
<tr>
<td>traversing a tree</td>
<td>tree with ( n ) nodes</td>
<td>accessing a node</td>
</tr>
<tr>
<td>Towers of Hanoi</td>
<td>( n ) disks</td>
<td>moving a disk</td>
</tr>
</tbody>
</table>
**Growth Rate (Cont.)**

**Best, Worst and Average Cases:**

The number of basic operations performed by an algorithm usually varies for inputs of a given size. Thus, we use the concepts of **best-case, worst-case, and average complexity** of an algorithm.

Let \( \Psi n \) denote the set of all inputs of size \( n \) to an algorithm and \( \tau(I) \) denotes the number of basic operations that are performed when the algorithm is executed with input \( I \), which varies over all inputs of size \( n \).

**Defn:** **Best-case complexity** of an algorithm is the function \( B(n) \) such that \( B(n) \) equals the minimum value of \( \tau(I) \). That is,

\[
B(n) = \min \{ \tau(I) | I \in \Psi n \}
\]

**Defn:** **Worst-case complexity** of an algorithm is the function \( W(n) \) such that \( W(n) \) equals the maximum value of \( \tau(I) \). That is,

\[
W(n) = \max \{ \tau(I) | I \in \Psi n \}
\]

**Defn:** **Average complexity** of an algorithm with finite set \( \Psi n \) is defined as

\[
A(n) = \sum_{I \in \Psi n} \tau(I) \cdot p(I)
\]

Average complexity \( A(n) \) is defined as the expected number of basic operations performed. \( p(I) \) is the probability of occurring each \( I \in \Psi n \) as the input to the algorithm. Alternatively, \( A(n) = (t1+\ldots+tg(n))/g(n) \), \( (t1+\ldots+tg(n)) \) covers all different cases.
Growth Rate (Cont.)

We shall usually concentrate on finding only the worst-case running time \(W(n)\), that is, the longest running time for any input of size \(n\). The reasons for that:

- The worst-case running time of an algorithm is an upper bound on the running time for any input. Algorithm will take longer (we hope).
- For some algorithms, the worst-case occurs fairly often, e.g., searching databases, absent information.
- The average case is often roughly as bad as the worst case.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(B(n))</th>
<th>(A(n))</th>
<th>(W(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Search</td>
<td>1</td>
<td>(n)</td>
<td>(n)</td>
</tr>
<tr>
<td>BinarySearch</td>
<td>1</td>
<td>(\log n)</td>
<td>(\log n)</td>
</tr>
<tr>
<td>Max,Min,MaxMin</td>
<td>(n)</td>
<td>(n)</td>
<td>(n)</td>
</tr>
<tr>
<td>InsertionSort</td>
<td>(n)</td>
<td>(n^2)</td>
<td>(n^2)</td>
</tr>
<tr>
<td>QuickSort</td>
<td>(n\log n)</td>
<td>(n\log n)</td>
<td>(n^2)</td>
</tr>
<tr>
<td>Towers</td>
<td>(2^n)</td>
<td>(2^n)</td>
<td>(2^n)</td>
</tr>
</tbody>
</table>

Figure: Order of best-case, average, and worst-case complexities
Growth Rate (Cont.)

Classes of algorithms that arise from the dominant terms:

Logarithmic algorithms: Algorithms whose efficiency is dominated by $\log_a n$ term

Polynomial algorithms: Algorithms whose efficiency is expressed in terms of polynomial form, categorized by $n^m$ (linear, quadratic, cubic, etc.)

Exponential algorithms: Algorithms whose efficiency is dominated by a term of the form $a^n$

Figure: Rate of growth of common computing time functions
Importance of developing Efficient Algorithms:

Sequential search vs Binary search

<table>
<thead>
<tr>
<th>Array size</th>
<th>No. of comparisons by seq. src</th>
<th>No. of comparisons by bin. Srch</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>128</td>
<td>8</td>
</tr>
<tr>
<td>1,048,576</td>
<td>1,048,576</td>
<td>21</td>
</tr>
<tr>
<td>~4.10^9</td>
<td>~4.10^9</td>
<td>33</td>
</tr>
</tbody>
</table>

Execution times for algorithms with the given time complexities:

<table>
<thead>
<tr>
<th>n</th>
<th>f(n) = n</th>
<th>nlgn</th>
<th>n^2</th>
<th>2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.02 μs</td>
<td>0.086 μs</td>
<td>0.4 μs</td>
<td>1 ms</td>
</tr>
<tr>
<td>10^6</td>
<td>1μs</td>
<td>19.93 ms</td>
<td>16.7 min</td>
<td>31.7 years</td>
</tr>
<tr>
<td>10^9</td>
<td>1s</td>
<td>29.9s</td>
<td>31.7 years</td>
<td>! centuries</td>
</tr>
</tbody>
</table>

Figure: Algorithms versus hardware
Examples

An Example: Sorting Problem

**Input:** A sequence of n numbers \(<a_1, a_2, \ldots, a_n>\).

**Output:** A permutation (reordering) \(<a'_1, a'_2, \ldots, a'_n>\) of the input sequence such that \(a'_1 \leq a'_2 \leq \ldots \leq a'_n\).

An example sorting algorithm: *Insertion sort:*

Insertion-Sort uses an *incremental* design approach: having sorted the subarrays \(A[1..j-1]\), we insert the singles element \(A[j]\), into its proper place, yielding the sorted subarray \(A[1..j]\).

Array \(A[1..n]\) contains a sequence of length \(n\) that is to be sorted.

An example:

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

‘\_\_’ represents the position of index \(j\).
**Insertion-Sort (A)**

<table>
<thead>
<tr>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_1 )</td>
<td>( n )</td>
</tr>
<tr>
<td>( c_2 )</td>
<td>( n-1 )</td>
</tr>
<tr>
<td>( c_4 )</td>
<td>( n-1 )</td>
</tr>
<tr>
<td>( c_5 )</td>
<td>( \sum_{2&lt;j&lt;n} t_j )</td>
</tr>
<tr>
<td>( c_6 )</td>
<td>( \sum_{2&lt;j&lt;n} (t_j-1) )</td>
</tr>
<tr>
<td>( c_7 )</td>
<td>( \sum_{2&lt;j&lt;n} (t_j-1) )</td>
</tr>
<tr>
<td>( c_8 )</td>
<td>( n-1 )</td>
</tr>
</tbody>
</table>

// sort in increasing order //

1. \textbf{for} \( j \leftarrow 2 \) \textbf{to} length[A]
2. \quad \textbf{do} key \leftarrow A[j]
3. \quad \quad // insert A[j] into the sorted sequence A[1..j-1] //
4. \quad \quad i \leftarrow j-1
5. \quad \quad \textbf{while} i > 0 and A[j] > key \quad \quad \quad \quad c_5 \quad \quad \sum_{2<j<n} t_j
6. \quad \quad \quad \quad \textbf{do} A[i+1] \leftarrow A[i]
7. \quad \quad \quad \quad i \leftarrow i-1
8. \quad \quad A[i+1] \leftarrow key

**Analysis of Insertion sort algorithm:**

\( T(n) \), the running time of insertion sort is the sum of products of the cost and times.

\[
T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5\sum_{2<j<n} (t_j) + c_6\sum_{2<j<n} (t_j-1) + c_7\sum_{2<j<n} (t_j-1) + c_8(n-1)
\]

\( t_j = 1 \) for \( j = 2,..,n \) for the \textit{best} case, array is already sorted,

\[
T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1+c_2+c_4+c_5+c_8)n - (c_1+c_2+c_4+c_5+c_8), \text{ that is, } T(n) \Theta(n)
\]

\( t_j = j \) for \( j = 2,..,n \) for the worst case, array is reverse sorted,

\[
\sum_{2<j<n} j = (n(n+1)/2) - 1 \text{ and } \sum_{2<j<n} j-1 = n(n-1)/2
\]

\[
T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5((n(n+1)/2)-1) + c_6(n(n-1)/2) + c_7(n(n-1)/2) + c_8(n-1)
\]

\[
= (c_5/2+c_6/2+c_7/2)n^2 + (c_1+c_2+c_4+c_5/2-c_6/2-c_7/2+c_8)n - (c_2+c_4+c_5+c_8), \text{ which is } \Theta(n^2)
\]
Computational Complexity of Algorithms

Here we introduce several types of asymptotic notations. The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers \( \mathbb{N} = \{0,1,2,\ldots\} \).

**Θ-Notation:** If \( f(n) \) and \( g(n) \) are running time functions, then we say that \( f(n) \in \Theta(g(n)) \) if there exists positive real constants \( c_1 \) and \( c_2 \), and a positive integer \( n_0 \), such that

\[
     c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)
\]

for all \( n \geq n_0 \). That is, \( f(n) \) is bounded from above and below by \( g(n) \). That is, Θ-notation bounds a function within constant factors. \( f(n) \) always lies between \( c_1 \cdot g(n) \) and \( c_2 \cdot g(n) \) inclusive.
Computational Complexity of Algorithms

Example: We want to show that \( \frac{1}{2}n^2 - 3n = \Theta(n^2) \).

Soln: \( f(n) = \frac{1}{2}n^2 - 3n \quad g(n) = n^2 \)

To show desired result, we need determine positive constants \( c_1, c_2, \) and \( n_0 \) such that

\[
0 \leq c_1 \cdot n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 \cdot n^2 \quad \text{for all} \quad n \geq n_0.
\]

Dividing by \( n^2 \), we get

\[
0 \leq c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2
\]

\( c_1 \leq \frac{1}{2} - \frac{3}{n} \) holds for any value of \( n \geq 7 \) by choosing \( c_1 \leq 1/14 \)

\( \frac{1}{2} - \frac{3}{n} \leq c_2 \) holds for any value of \( n \geq 1 \) by choosing \( c_2 \geq \frac{1}{2} \).

Thus, by choosing \( c_1 = 1/14 \) and \( c_2 = \frac{1}{2} \) and \( n_0 = 7 \), we can verify \( \frac{1}{2}n^2 - 3n = O(n^2) \).

Certainly other choices for the constants exist.

Properties of \( \Theta \)-notation

- \( f(n) \) is \( \Theta(f(n)) \) (reflexivity)
- \( f(n) \) is \( \Theta(g(n)) \) iff \( g(n) \) is \( \Theta(f(n)) \) (symmetry)
- If \( f(n) \) is \( \Theta(g(n)) \) and \( g(n) \) is \( \Theta(h(n)) \) then \( f(n) \) is \( \Theta(h(n)) \) (transitivity)
- For any \( c > 0 \), the function \( c \cdot f(n) \) is \( \Theta(f(n)) \)
- If \( f_1 \) is \( \Theta(g(n)) \) and \( f_2 \) is \( \Theta(g(n)) \), then \( (f_1 + f_2)(n) \) is \( \Theta(g(n)) \)
- If \( f_1 \) is \( \Theta(g_1(n)) \) and \( f_2 \) is \( \Theta(g_2(n)) \), then \( (f_1 \cdot f_2)(n) \) is \( \Theta(g_1 \cdot g_2(n)) \)
**Comp. Complexity of Alg. (cont.)**

**O-Notation:** Given two running time functions \( f(n) \) and \( g(n) \), we say that \( f(n) \) is \( O(g(n)) \) if there exists a real constant \( c \geq 0 \), and a positive integer \( n_0 \), such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \).

O-notation gives an upper bound for a function to within a constant factor.

**Example:** We want to show that \( 1/2n^2 + 3n \in O(n^2) \)

\[
\begin{align*}
\text{f(n) = } & \frac{1}{2}n^2 + 3n \\
\text{g(n) = } & n^2
\end{align*}
\]

to show desired result, need \( c \) and \( n_0 \) such that \( 0 \leq \frac{1}{2}n^2 + 3n \leq c \cdot n^2 \)

try \( c = 1 \)

\[
\begin{align*}
\frac{1}{2}n^2 + 3n & \leq n^2 \\
3n & \leq \frac{1}{2}n^2 \\
6 & \leq n
\end{align*}
\]

i.e. \( n_0 = 6 \).
**Comp. Complexity of Alg. (Cont.)**

**Ω- Notation:** We say that $f(n)$ is $\Omega(g(n))$ if there exists a real constant $c \geq 0$, and positive integer $n_0$, such that

$$c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0$$

**Example:** We show that $n^2 + 10n \in \Omega(n^2)$. Because for $n \geq 0$, $n^2 + 10n \geq n^2$, we can take $c = 1$ and $n = 0$ to obtain the result.

![Graph showing relationships between f(n), cg(n), and f(n) ∈ Ω(g(n)).]

**Relationships among O, Ω, and Θ - Notations**

1. $f(n)$ is $O(g(n))$ iff $g(n)$ is $Ω(f(n))$
2. $f(n)$ is $Θ(g(n))$ iff $f(n)$ is $O(g(n))$ and $f(n)$ is $Ω(g(n))$,
3. $Θ(g(n)) = O(g(n)) \cap Ω(g(n))$,
O-Notation (for two variables): Given two real-valued functions $g(n,m)$ and $f(n,m)$ of two nonnegative variables $n$ and $m$, we say that $g(n,m)$ and $f(n,m)$ are eventually nonnegative if there exists a positive number $n_0$ such that $g(n,m) \geq 0$ whenever $n \geq n_0$ and $m \geq n_0$. We say that $f(n,m)$ is $O(g(n,m))$ if the set $O(g(n,m))$ consists of those eventually nonnegative functions $f(n,m) \in O(g(n,m))$ if, and only if, there exists positive integers $c$ and $n_0$, such that

$$f(n,m) \leq c \cdot g(n,m) \text{ for all } n,m \geq n_0.$$ 

The sets $\Theta(g(n,m))$ and $\Omega(g(n,m))$ are similarly defined. The notions also extend to functions of more than two variables.
Comp. Complexity of Alg. (cont.)

**o- Notation:** f(n) is o(g(n)), “little-oh of g of n” as the set o(g(n)) = \{f(n) : for any positive constant c > 0, there exists a constant n_0 \geq 0 such that
- 0 \leq f(n) < c.g(n) for all n \geq n_0\}
- We use o-notation to denote an upper bound that is not asymptotically tight, whereas O-notation may be asymptotically tight. Intuitively, in the o-notation, the function f(n) becomes insignificantly relative to g(n) as n approaches infinity, that is,
\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0
\]
*Example:* 2n = o (n^2), but 2n^2 \neq o(n^2)
*Example:* n^2 /2 \in o(n^3), because \[
\lim_{n \to \infty} \frac{(n^2 /2)}{n^3} = \lim_{n \to \infty} \frac{1}{2n} = 0
\]
*Proposition:* f(n) \in o(g(n)) \Rightarrow O(f(n)) \subset O(g(n))

**ω - Notation:** f(n) is ω(g(n)), “little-omega of g of n” as the set ω(g(n)) = \{f(n) : for any positive constant c > 0, there exists a constant n_0 \geq 0 such that
- 0 \leq c.g(n) < f(n) for all n \geq n_0\}
ω-notation denotes a lower bound that is not asymptotically tight. The relation f(n) = ω(g(n)) implies that,
\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty, \text{ if the limit exists.}
\]
That is, f(n) becomes arbitrarily large relative to g(n) as n approaches infinity.

*Example:* n^2/2 = ω(n), but n^2/2 \neq ω(n^2)
Example: \( n^2/2 = \omega(n) \), but \( n^2/2 \neq \omega(n^2) \)

Example: Factorials, \( n! \), \( n \geq 0 \)

\[
\begin{align*}
n! & = 1 \quad \text{if } n = 0, \\
n^*(n-1)! & \quad \text{if } n > 0
\end{align*}
\]

- A weak upper bound on the factorial function is \( n! \leq n^n \), since each \( n \) terms in the factorial product is at most \( n \).
- Stirling’s approximation is \( n! = \sqrt{2\pi n} \ (n/e)^n \ (1+\Theta(1/n)) \), which gives a tighter upper bound, and a lower bound as well. Using Stirling’s approximation, one can prove

\[
n! = o(n^n), \quad n! = \omega(2^n), \quad \lg(n!) = \Theta(n \lg n)
\]

The following bounds also hold for all \( n \):

\[
\sqrt{2\pi n} \ (n/e)^n \leq n! \leq \sqrt{2\pi n} \ (n/e)^{n+(1/12n)}
\]

**Definition:** Given the function \( g(n) \), we define \( \sim(g(n)) \) to be the set of all functions \( f(n) \) having the property that

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1,
\]

If \( f(n) \in \sim(g((n))) \), then we say that \( f(n) \) is strongly asymptotic to \( g(n) \) and denote this by writing \( f(n) \sim g(n) \).

**Property:** \( f(n) \sim g(n) \Rightarrow f(n) \in \Theta(g(n)) \)
Definition: A function $f(n)$ is said to grow slower than a function $g(n)$ (or, $g(n)$ grows faster than $f(n)$), denoted by $f(n) \leq \text{g}(n)$, where the relation $\leq$ is a partial ordering, if $\lim_{n \to \infty} f(n)/g(n) = 0$. For example, $g(n) = n^2$ grows faster than $f(n) = n$ since $\lim_{n \to \infty} n/ n^2 = \lim_{n \to \infty} 1/ n = 0$.

We write $f(n) = o(g(n))$ if $f(n) \leq g(n)$, and $f(n) = O(g(n))$ if for some constant $K$ and almost all $n \geq 0$ (for all, but finitely many $n \geq 0$), $f(n) \leq Kg(n)$.

The following sequences of functions appear very often in the study of computational complexity:

- Poly-log sequence: $\{(\log n)^i \mid i = 1, 2, \ldots\}$
- Polynomial sequence: $\{n^i \mid i = 1, 2, \ldots\}$
- Subexponential sequence: $\{n^{(\log n)^i} \mid i = 1, 2, \ldots\}$
- Exponential sequence: $\{2^{ni} \mid i = 1, 2, \ldots\}$
- Superexponential sequence: $\{2^{ni} \mid i = 1, 2, \ldots\}$
These sequences satisfy the following properties:

- In each sequence, if \( i < j \), then \( i^{th} \) function grows slower than the \( j^{th} \) function. For instance, \( n(\log n)^3 \succ n(\log n)^4 \).
- For each sequences, every function in the former sequence grows slower than any function in the latter sequence (except for the first function of the last sequence). For instance, \( \log n^6 \succ n^{10} \succ n(\log n)^4 \succ 2^{3n} \succ 2n^2 \).

These five sequences do not contain all possible measurement for growth rates. There are other examples not in these sequences.

- For example, the function \( T(n) = 2^\sqrt{\log n} \) grows slower than every function in the polynomial sequence, but faster than every function in the poly-log sequence. That is, \( (\log n)^i \succ 2^\sqrt{\log n} \succ n \).

We can show this by the following:

\[
\lim_{n \to \infty} (\log n)^i / 2^\sqrt{\log n} = 0 \quad \text{and} \quad \lim_{n \to \infty} 2^\sqrt{\log n} / n = 0
\]
**Theorem:** Let the polynomial

\[ P(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0 \]

be degree of \( k, a_k > 0 \). Then, \( P(n) \) is \( \Theta(n^k) \).

**Proof:** It is sufficient to show that \( P(n) \in O(n^k) \) and \( P(n) \in \Omega(n^k) \). We first show that \( P(n) \in O(n^k) \).

\[ P(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0 \]

\[ \leq a_k n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| n^1 + |a_0| n^0 \]

Thus, if \( c = (|a_0| + |a_1| + |a_2| + \cdots + |a_{k-1}| + a_k) \), then \( P(n) \leq c n^k \) for all \( n \geq n_0 = 1 \), so that \( p(n) \in O(n^k) \). We now show that \( P(n) \in \Omega(n^k) \).

\[ P(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_{k-1} n^{k-1} + a_k n^k \]

\[ \geq a_k n^k - |a_{k-1}| n^{k-1} - \cdots - |a_1| n^1 - |a_0| n^0 = (a_k/2) n^k + [(a_k/2) n^k - |a_{k-1}| n^{k-1} - \cdots - |a_2| n^2 - |a_1| n - |a_0|] \]

\[ \geq (a_k/2) n^k + [(a_k/2) n^k - |a_{k-1}| n^{k-1} - \cdots - |a_2| n^2 - |a_1| n - |a_0| n^{k-1}] \]

\[ = (a_k/2) n^k + [(a_k/2) n - (|a_{k-1}| + \cdots + |a_2| + |a_1| + |a_0|)] n^{k-1} \]

Thus, if \( c = a_k/2 \) and \( n_0 = (|a_{k-1}| + \cdots + |a_2| + |a_1| + |a_0|))(2/a_k) \), the term inside the bracket in the last equation is nonnegative for all \( n \geq n_0 \), and hence \( P(n) \geq c n^k \) for all \( n \geq n_0 \). This shows that \( P(n) \in \Omega(n^k) \).

Therefore, \( P(n) \) is \( \Theta(n^k) \).
Proposition: \( P(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \) be any polynomial of degree of \( k, a_k > 0 \). Then, 
\[ P(n) \sim a_k n^k \]

Proof: Consider the ratio \( \frac{a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0}{a_k n^k} \) as \( n \) tends to infinity:
\[
\lim_{n \to \infty} \left( \frac{a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0}{a_k n^k} \right) = \lim_{n \to \infty} \left( 1 + \frac{a_{k-1}}{a_k n} + \ldots + \frac{a_1}{a_k n^{k-1}} + \frac{a_0}{a_k n^k} \right) = 1.
\]

Proposition: For any nonnegative real constants \( k \) and \( a \), with \( a > 1 \), \( O(n^k) \subset O(a^n) \).

Proof: Consider the ratio \( \frac{n^k}{a^n} \). Repeated applications of L’Hôpital’s Rule yields
\[
\lim_{n \to \infty} \frac{n^k}{a^n} = \lim_{n \to \infty} \frac{k n^{k-1}}{a^n (\ln a) n}
= \lim_{n \to \infty} \frac{k(k-1) n^{k-2}}{a^n (\ln a)^2 n}
\ldots
\ldots
= \lim_{n \to \infty} \frac{k!}{(\ln a)^k a^n}
= 0.
\]

Hence, \( n^k \in o(a^n) \).
Since, if \( f(n) \in o(g(n)) \Rightarrow O(f(n) \subset O(g(n)) \), it is true that \( O(n^k) \subset O(a^n) \).
Proposition: \( L(n) = \log(n!) \in \Theta(n \log n) \).

Proof: \( L(n) = \log 1 + \log 2 + \ldots + \log n \).

Clearly, \( L(n) \in O(n \log n) \), since

\[ L(n) = \log 1 + \log 2 + \ldots + \log n \leq \log n + \log n + \ldots + \log n = n \log n, \]

So that \( L(n) \in O(n \log n) \).

It remains to show that \( L(n) \in \Omega(n \log n) \). Let \( m = \lfloor n/2 \rfloor \). We have

\[ L(n) = (\log 1 + \log 2 + \ldots + \log m) + [\log (m + 1) + \log (m+2) + \ldots + \log n], \]

\[ \geq \log (m + 1) + \log (m+2) + \ldots + \log n \]

\[ \geq \log (m + 1) + \log (m+1) + \ldots + \log (m+1) \]

\[ = (n-m) \log (m+1) \]

\[ \geq (n/2) \log (n/2) = (n/2) (\log n - \log 2) \]

Clearly, \( (n/2) (\log n - \log 2) \geq (n/2) (\log n - (1/2) \log n) \) for all \( n \geq n_0 \), where \( n_0 \) is sufficiently large (the constant \( n_0 \) depends on the base chosen).

Thus, \( L(n) \geq (1/4) n \log n \), so that \( L(n) \in \Omega(n \log n) \).

Therefore, \( \log(n!) \in \Theta(n \log n) \).
**Theorem**: The function \( h(n) = \sum_{1 \leq i \leq n} \frac{1}{i} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \) is \( P(n) \) is \( \Theta(\log n) \).

**Proof**: Values \( c_1, c_2, \) and \( n_0 \) can be arrived at by finding upper and lower bounds for \( h(n) \) using only terms of the form \( 1/2^j \) as follows:

\[
1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \ldots + \frac{1}{2^k}
\]

\[
\leq 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{2^k} = h(2^k)
\]

\[
\leq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^k}
\]

The first and third sums can be evaluated and inequalities become:

\[
1 + \frac{k}{2} \leq h(2^k) \leq k + \frac{1}{2^k}
\]

For an integer \( n \), let \( k = \lfloor \log n \rfloor \), \( 2^k = n \); then

\[
1 + \frac{\log n}{2} \leq h(n) \leq \log n + \frac{1}{n},
\]

If we use the following inequilities: \( \log n > 1/n \) and \( \log n / 2 < 1 + k/2 \), then

\[
\frac{1}{2}(\log n) \leq h(n) \leq 2(\log n)
\]

Therefore, \( h(n) = \Theta(\log n), c_1 = \frac{1}{2}, c_2 = 2, n_0 = 1. \)
Arithmetic sequence: \( S(n) = \sum_{1 \leq i \leq n-1} a + id = na + d \sum_{1 \leq i \leq n-1} i, \)
where \( a \) and \( d \) are real constants; \( d \) is called the difference of the sequence, since consecutive terms differ by \( d \). If we solve this, we find \( S(n) = na + d.n.(n-1)/2 \). Here \( S \) is a polynomial of degree 2, which implies that \( S \) is \( \Theta(n^2) \).

Geometric sequence: \( G(n) = \sum_{0 \leq i \leq n-1} ar^i = a + ar + \ldots + ar^{n-1} \) where \( r \) is a real constant, called the ratio of the sequence. Multiplying both sides by \( r \),
\[
 rG = \sum_{0 \leq i \leq n-1} ar^{i+1} = ar + ar^2 + \ldots + ar^n = -a + a + ar + ar^2 + \ldots + ar^n
\]
\[
 rG = G + ar^n - a
\]
\[
 G = (ar^n - a)/(r-1) = a(r^n - 1) / (r-1) \text{ if } r \neq 1.
\]
Note that \( G = an \) if \( r = 1 \).

In conclusion
\( G \) is \( \Theta(n) \) if \( r = 1 \)
\( G \) is \( \Theta(r^n) \) if \( r \neq 1 \)
\( G \) is \( O(1) \) if \( r < 1 \)
Comp. Complexity of Alg. (cont.)

Growth rates of some other functions

1. \( f(n) = \sum_{1 \leq i \leq n} i = n(n+1)/2 \)
2. \( f(n) = \sum_{1 \leq i \leq n} i^2 = n(n+1)(2n+1)/6 \)
2’. \( f(n) = \sum_{1 \leq i \leq n} i^k = \left[\frac{1}{(k+1)}\right] (n^{k+1}) \)
3. \( f(n) = \sum_{0 \leq i \leq n} x^i = (x^{n+1} - 1)/(x - 1) \)
4. \( f(n) = \sum_{0 \leq i \leq n} 2^i = (2^{n+1} - 1)/(2 - 1) = 2^{n+1} - 1 \)

Sample Algorithm analysis:

The sum rule:
Given the \( a \) functions \( f_1, f_2, \ldots, f_a \), where \( a \) is constant, if \( f_i = O(g_i), \forall i \leq n \), then
\[
\sum_{1 \leq i \leq a} f_i = O(g_m)
\]
where \( g_m \) is the fastest growing of the functions \( g_1, g_2, \ldots, g_a \).

The product rule:
If \( f_i = O(g_i), \forall i \leq n \), where \( a \) is constant, then
\[
\prod_{1 \leq i \leq a} f_i = O(\prod_{1 \leq i \leq a} g_i)
\]
Comp. Complexity of Alg.(cont.)

Example:
For k = 1 to n/2 do
{ 
    ....
}
For j = 1 to n*n do
{ 
    ----
}
\[ \sum_{k=1}^{n/2} c + \sum_{j=1}^{n*n} c = c.n/2 + c.n^2 = O(n^2) \]

Example:
For k = 1 to n/2 do
{ 
    For j = 1 to n*n do
    { 
        ----
    }
}
\[ \sum_{k=1}^{n/2} \sum_{j=1}^{n*n} c = c.n/2 * n^2 = O(n^3) \]
Comp. Complexity of Alg.(cont.)

Example:
    i = n
    while i > 1 do
    {
    i = i div 2
    }

\[ \sum_{i=2}^{n} (1/2^i) = O(\log_2 n) \]

Example:
For i = 1 to n do
    For j = i to n do
        For k = i to j do
            \[ m = m + i + j + k \]

\[ \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 3 \approx n^3/10 \text{ additions, that is } O(n^3) \]

\[ \sum_{i=1}^{n} \sum_{j=i}^{n} 3(j-i+1) = \sum_{i=1}^{n} [3 \sum_{j=i}^{n} j - 3 \sum_{j=i}^{n} i + 3 \sum_{j=i}^{n} 1] \]
\[ = \sum_{i=1}^{n} [3(\sum_{j=1}^{n} j - \sum_{j=1}^{i-1} j) - 3i(n-i+1) + 3(n-i+1)] \]
\[ = \sum_{i=1}^{n} [3[(n(n+1))/2 - (i(i-1))/2] - 3i(n-i+1) + 3(n-i+1)] \]

\[ \approx n^3/10 \text{ additions, that is, } O(n^3) \]
Recurrences

There are four methods for solving recurrences, which is for obtaining asymptotic $\Theta$ and $O$ bounds on the solution.

1. The iteration method converts the recurrence into a summation and then relies on techniques for bounding summations to solve the recurrence.
2. In the substitution method, we guess a bound and then use mathematical induction to prove our guess correct.
3. The master method provides bounds for recurrences of the form $T(n) = aT(n/b) + f(n)$ where $a \geq 1$, $b \geq 2$, and $f(n)$ is a given function.
4. Using the characteristic equation.
Recurrences (cont.)

2 Solving recurrences by iteration method: converts the recurrence into a summation and then relies on techniques for bounding summations to solve the recurrence.

Example: The recurrence arises for a recursive program that makes at least $n$ number of disk moves from one peg to another peg and then move them to the third one.

$$t_n = 2t_{n-1} + 1, \quad n \geq 2, \quad \text{subject to } t_1 = 1.$$ 

$$t_n = 2t_{n-1} + 1 = 2^2t_{n-2} + 2 + 1 = \ldots = 2^{n-1}t_1 + 2^{n-2} + \ldots + 2 + 1$$ 

$$= 2^n - 1. \quad \Rightarrow \quad t_n \in O(2^n).$$

- This is the minimum number of moves required to transfer $n$ disks from one peg to another. Indeed, before this move, we must first move all other disks to the third disk and also, after the move of the largest disk, we must move all other disks to the top of the largest disk.

- Therefore, the minimum number $f(n)$ of moves satisfies the following inequality:

$$f_1 = 1 \quad \text{and} \quad f_n \geq 2f_{n-1} + 1, \quad n \geq 2.$$ 

Thus, $f_n \geq t_n = 2^n - 1 \Rightarrow f_n \in O(2^n)$

For $n = 64$, it takes $584 \times 10^6$ years if every 1000 moves takes 1 seconds.
Recurrences (cont.)

2 Example: The recurrence arises for a recursive program that loops through the input to eliminate one item:
\[ t_n = t_{n-1} + n \quad \text{for } n \geq 2, \text{ subject to } t_1 = 1. \]

To solve such a recurrence, we “telescope” it by applying it to itself, as follows:
\[
\begin{align*}
    t_n &= t_{n-1} + n \\
    t_n &= t_{n-2} + (n-1) + n \\
    t_n &= t_{n-3} + (n-2) + (n-1) + n \\
    \quad \vdots \\
    t_n &= t_1 + 2 + \ldots + (n-2) + (n-1) + n \\
    t_n &= 1 + 2 + \ldots + (n-2) + (n-1) + n \\
    &= n(n+1)/2 \\
\end{align*}
\]

Hence, \( t_n \in \Theta(n^2/2) \)
Recurrences (cont.)

Example: The recurrence arises for a recursive program that has to make a linear pass through the input, before, during, or after it is split into two halves:

\[ t_n = 2t_{n/2} + n \quad \text{for} \quad n \geq 2, \text{ subject to } t_1 = 0. \]

To solve such a recurrence, we assume that \( n = 2^k \) (\( k = \log n \)) and then “telescope” it by applying it to itself, as follows:

\[
\begin{align*}
  t_n &= 2(2t_{n/4} + n/2) + n \\
  &= 4t_{n/4} + 2n/2 + n = 4t_{n/4} + n (1+1) \\
  &= 4(2t_{n/8} + n/4) + 2n/2 + n \\
  &= 8t_{n/8} + 4n/4 + 2n/2 + n = 8t_{n/8} + n (1+1+1) \\
  & \quad \vdots \\
  &= nt_{n/n} + n(1+\ldots+1) \\
  &= 2^k t_1 + n.k \\
  t_n &= 0 + n(\log n) \\
  \text{Hence, } t_n &\in \Theta(n\log n)
\end{align*}
\]
Recurrences (cont.)

2 Solving recurrences by substitution (guess) method:
Guesses a bound and then use mathematical induction to prove the guess correct.

Example: The recurrence arises for a recursive program that halves the input in one step:

\[ t_n = t_{n/2} + c \] \( n \geq 2, \) subject to \( t_1 = 1. \)
\[ t_2 = 1 + c \]
\[ t_4 = 1 + 2c, \; t_8 = 1 + 3c, \; \ldots. \]
\[ t_k = 1 + kc, \text{ where } n = 2^k, \; t_n = 1 + c \cdot \log n, \text{ Therefore, } t_n \in \Theta(\log n) \]

Proof by induction:

Base case: \( n=1, \; t_1 = 1 \) is given

Induction case: \( n=k, \; t_k = 1 + kc, \) where \( n = 2^k, \) for \( t_k = t_{k/2} + c, \) is true

Proof: \( n=k+1, \; t_{k+1} = 1 + (k+1)c = 1+ kc + c \) ??

From the given recurrence

\( (t_{k+1} = T(2^{k+1}) = T \left( \frac{2^{k+1}}{2} \right) + c = T(2^k) + c = t_k + c = 1+kc + c \) (by the induction step).

Therefore, \( t_n = 1 + c \cdot \log n, \) for \( n = 2^k \)
Recurrences (cont.)

- **Example:** The recurrence arises for a recursive program that halves the input in one step, applies to 3 subproblems and loops through each input:
  - $t_n = 3t_{n/2} + cn \quad n \geq 2, \text{ subject to } t_1 = 1.$
  - $t_2 = 3 + 2c$
  - $t_4 = 9 + 10c$
  - $t_8 = 27 + 38c$
  - $t_{16} = 81 + 130c$
  - $t_{2^k} = 3^k + 2^k c \left[ \frac{(3/2)^k - 1}{(3/2) - 1} \right], \text{ where } n = 2^k \text{ and } k = \lg n$
  - $t_n = 3^{\lg n} + c n \left( 3^{\lg n/n} - 1 \right) / 1/2$
  - $t_n = n^{1.59} + 2cn \left( n^{0.59} - 1 \right), \text{ where } 3^{\lg n} = n^{\lg 3} = 3^{1.59}$
  - $= n^{1.59} (1 + 2c) - 2cn$
  - $t_n \in \Theta(n^{1.59})$
Recurrences (cont.)

3 Solving recurrences by master method:
The master method depends on the following theorem:

**Master theorem:** Let \( a \geq 1 \) and \( b \geq 2 \) be constants and let \( T(n) \) be a function, and let \( T(n) \) be defined on the nonnegative integers by the recurrence
\[
T(n) = aT(n/b) + cn^i \quad (n > n_0), \text{ when } n/ n_0 \text{ is a power of } b.
\]

Show that the exact order of \( T(n) \) is given by

\[
T(n) \in \begin{cases} 
\Theta(n^{\log_b a}) & \text{case 1: } a = b^i, i = \log_b a \\
\Theta(n^{\log_b a}) & \text{case 2: } a > b^i, i < \log_b a \\
\Theta(n^i) & \text{case 3: } a < b^i, i > \log_b a
\end{cases}
\]

**Example:** The recurrence arises for a recursive program that halves the input, but perhaps must examine every item in the input:
\[
t_n = t_{n/2} + n \quad \text{for } n \geq 2, \text{ subject to } t_1 = 0.
\]

For this recurrence, \( a = 1, b = 2, i = 1 \), and thus case 3: \( a < b^i \) applies and
\[
t_n \in \Theta(n)
\]

**Example:** The recurrence arises from matrix multiplication:
\[
t_n = 8t_{n/2} + cn^2 \quad \text{for } n \geq 2, \text{ subject to } t_1 = 1.
\]

For this recurrence, \( a = 8, b = 2, i = 2 \), and thus case 3: \( a > b^i \) applies and
\[
t_n \in \Theta(n^{\log_2 8}) = \Theta(n^3)
\]
Recurrences (cont.)

**Master Theorem (Another Version (text book)):** let $a \geq 1$ and $b>1$ be constants, let $f(n)$ be a function, and $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n), \ a \geq 1 \text{ and } b > 1.$$

The function divides the problem of size $n$ into a subproblems, each of size $n/b$. The subproblems are solved recursively each in time $T(n/b)$. The cost of dividing the problem and combining the results of the subproblems is described by the $f(n)$. We interpret $n/b$ to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$, which does not affect the asymptotic behaviour of the recurrence.

$T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$.

Intuitively, the solution to the recurrence is determined by the larger of the two functions, $f(n)$ and $n^{\log_b a}$. In case 1, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be polynomially smaller. That is, $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of $n^\varepsilon$ for some constant $\varepsilon > 0$. Similar technicality is valid for case 3, in addition satisfy the “regularity” condition that $af(n/b) \leq cf(n)$. 

*Analysis of Algorithms, A.Yazici, Fall 2004*
The three cases do not cover all the possibilities for \( f(n) \).

**Example:** The recurrence arises for a recursive program that divides the input into three and solves 9 of sub problems, and perhaps must examine every item in the input:

\[
t_n = 9t_{n/3} + n, \text{ for } n \geq 3, \text{ subject to } t_1 = 1.
\]

Here \( a = 9, b = 3, f(n) = n \) and \( n^\log_a b = n^\log_3 9 = \Theta(n^2) \).

Since \( f(n) = \Theta(n^{\log_3 9 - \epsilon}) \), where \( \epsilon = 1 \), we can apply case 1 of the master theorem and conclude that the solution is \( t_n = \Theta(n^2) \).

**Example:** A recurrence arises from a search algorithm.

\[
t_n = t_{2n/3} + 1, \text{ for } n \geq 3, \text{ subject to } t_2 = 1.
\]

Here \( a = 1, b = 3/2, \) thus \( n^\log_a b = n^\log_{3/2} 1 = 1 \), and \( f(n) = 1 \).

Therefore, case 2 applies, since \( f(n) = \Theta(n^{\log_a b}) = \Theta(1) \), and the solution to the recurrence is \( t_n = \Theta(\log n) \).

**Example:** The recurrence arises from a recursive algorithm.

\[
t_n = 3t_{n/4} + \log n, \text{ for } n \geq 4, \text{ subject to } t_1 = 1.
\]

For this recurrence, \( a = 3, b = 4, \) and thus \( n^\log_a b = n^\log_4 3 = O(n^{0.793}) \), \( f(n) = \log n \).

Since \( f(n) = \Omega(n^{\log_4 3 + \epsilon}) \), where \( \epsilon \approx 0.2, \) case 3 applies if we can show that the regularity condition holds for \( f(n) \). For sufficiently large \( n \), \( af(n/b) = 3(n/4)\log(n/4) \leq (3/4)\log n = cf(n) \) for \( c = \frac{3}{4} < 0 \).

Consequently, by case 3, the solution to the recurrence is \( t_n \in \Theta(n\log n) \).

**Example:** The master method does not apply the following recurrence.

\[
t_n = 2t_{n/2} + \log n. \text{ In this } a = 2, b = 2, f(n) = \log n, \text{ and } n^\log_a b = n.
\]

It seems that case 3 should apply, since \( f(n) = \log n \) is asymptotically larger than \( n^\log_a b = n \) but not polynomially larger. The ratio \( f(n)/n^\log_a b = (\log n)/n = \log n \) is asymptotically less than \( n^\epsilon \) for any positive constant \( \epsilon \). Consequently, the recurrence falls into the gap between case 2 and case 3. Therefore, the master method cannot be used to solve this recurrence.
4 Solving Recurrences Using the Characteristic Equation

- **Homogeneous Recurrences:** First we introduce homogeneous linear recurrences with constant coefficients, that is, recurrence of the form

\[ a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0 \]  

(1)

where
- i. the \( t_i \) are the values we are looking for. The recurrence is linear since it does not contain terms of the form \( t_i t_{i+j}, t_i^2 \), and so on;
- ii. the coefficients \( a_i \) are constants; and
- iii. the recurrence is homogeneous because the linear combination of the \( t_i \) is equal to zero.

**Example:** Consider the familiar recurrence for the Fibonacci sequence.

\[ f_n = f_{n-1} + f_{n-2} \quad \text{which is} \quad f_n - f_{n-1} - f_{n-2} = 0, \]

Therefore, the Fibonacci sequence corresponds to a homogeneous linear recurrence with constants coefficients \( k = 2, a_0 = 1 \) and \( a_1 = a_2 = -1 \).

It is interesting to note that any linear combination of solutions is itself a solution.
Recurrences (cont.)

Trying to solve a few easy examples of recurrences of the form (1) by intelligent guesswork (or after a while intuition) may suggest we look for a solution of the form

\[ t_n = x^n \]

where \( x \) is a constant as yet unknown. If we try this solution in (1), we obtain

\[ a_0 x^n + a_1 x^{n-1} + \ldots + a_k x^{n-k} = 0 \]

This equation is satisfied if \( x = 0 \), a trivial solution of no interest. Otherwise, the equation is satisfied if and only if

\[ a_0 x^k + a_1 x^{k-1} + \ldots + a_k = 0. \]

This equation of degree \( k \) in \( x \) is called the characteristic equation of the recurrence of (1) and

\[ P(x) = a_0 x^k + a_1 x^{k-1} + \ldots + a_k \]

is called its characteristic polynomial.

Recall that the fundamental theorem of algebra states that any polynomial \( p(x) \) of degree \( k \) has exactly \( k \) roots (not necessarily distinct), which means that it can be factorized as a product of \( k \) monomials

\[ P(x) = \prod_{i=1}^{k} (x-r_i) \]

where the \( r_i \) may be complex numbers. Moreover, these \( r_i \) are the only solutions of the equation \( p(x) \).
Recurrences (cont.)

Suppose that the k roots $r_1, r_2, \ldots, r_k$ of this characteristic equation are all distinct. Since $p(r_i) = 0$, it follows that $x = r_i$ is a solution to the characteristic equation and therefore $r_i^n$ is a solution to the recurrence. Since any linear combination of solutions is also a solution, we conclude that the following satisfies the recurrence (1) for any choice of constants $c_1, c_2, \ldots, c_k$:

$$t_n = \sum c_i r_i^n$$

of terms $r_i^n$ is a solution of the recurrence (1), where the k constants $c_1, c_2, \ldots, c_k$ are determined by the initial conditions. We need exactly k initial conditions to determine the values of these k constants. The remarkable fact, which we do not prove here, is that (1) has only solutions of this form.

**Example**: Consider the recurrence

$$t_n - 3t_{n-1} - 4t_{n-2} = 0 \quad n \geq 2, \text{ subject to } t_0 = 0, t_1 = 1.$$ 

The characteristic equation of the recurrence is

$$x^2 - 3x - 4 = 0,$$ 

whose roots are $-1$ and $4$.

The general solution therefore has the form

$$t_n = c_1(-1)^n + c_24^n.$$ 

The initial conditions give

$$c_1 + c_2 = 0 \quad n = 0,$$

$$-c_1 + 4c_2 = 1 \quad n = 1, \text{ that is, } c_1 = -1/5, c_2 = 1/5$$

We finally obtain

$$t_n = 1/5 [4^n - (-1)^n], \text{ that is, } t_n \in \Theta(4^n).$$
**Recurrences (cont.)**

**Example:** (Fibonacci) Consider the recurrence
\[ t_n - t_{n-1} - t_{n-2} = 0 \quad n \geq 2, \text{ subject to } t_0 = 0, t_1 = 1. \]

The characteristic polynomial is
\[ P(x) = x^2 - x - 1 = 0 \]
Whose roots are \((1+\sqrt{5})/2\) and \((1-\sqrt{5})/2\).

The general solution is therefore of the form
\[ t_n = c_1((1+\sqrt{5})/2)^n + c_2((1-\sqrt{5})/2)^n. \]

The initial conditions give
\[ c_1 + c_2 = 0 \quad n = 0, \]
\[ ((1+\sqrt{5})/2)c_1 + ((1-\sqrt{5})/2)c_2 = 1 \quad n = 1, \]
Solving these equations, we obtain \(c_1 = 1/\sqrt{5}\), \(c_2 = -1/\sqrt{5}\).

We finally obtain
\[ t_n = 1/\sqrt{5} \left[ ((1+\sqrt{5})/2)^n - ((1-\sqrt{5})/2)^n \right], \]
which is de Moivre’s famous formula for the Fibonacci sequence.
Therefore, \(t_n \in \Theta((1+\sqrt{5})/2)^n\).
Recurrences (cont.)

Now suppose that the roots of the characteristic equation are not all distinct. Then
\[ t_n = nr^n \] is also a solution of (1). That is,
\[ a_0nr^n + a_1(n-1)r^{n-1} + \ldots + a_k(n-k)r^{n-k} = 0, \]
More generally, if \( m \) is the multiplicity of the root \( r \), then
\[ t_n = r^n, \quad t_n = nr^n, \quad t_n = n^2r^n, \ldots, \quad t_n = n^{m-1}r^n \]
are all possible solutions of (1).

The general solution is a linear combination of these terms and of the terms contributed by the other roots of the characteristic equation. Once again there are \( k \) constants to be determined by the initial conditions.

**Example**: Solve the following recurrence:
\[ t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3} \quad n \geq 3, \text{ subject to } t_0 = 0, t_1 = 1, \text{ and } t_2 = 2. \]

The characteristic equation is
\[ x^3 - 5x^2 + 8x - 4 = 0, \quad \text{or} \quad (x-1)(x-2)^2 = 0. \]
The roots are 1 and 2 (of multiplicity 2). The general solution is therefore
\[ t_n = c_1 1^n + c_2 2^n + c_3 n2^n. \]
The initial conditions give
\[ c_1 + c_2 = 0 \quad n = 0, \]
\[ c_1 + 2c_2 + 2c_3 = 1 \quad n = 1, \]
\[ c_1 + 4c_2 + 8c_3 = 2 \quad n = 2, \]
From which we find \( c_1 = -2, \ c_2 = 2, \ c_3 = -1/2. \) Therefore, \( t_n = 2^{n+1} - n2^{n-1} - 2. \)
\[ t_n \in \Theta(n2^n). \]
Recurrences (cont.)

Inhomogeneous Recurrences: We now consider recurrences of a slightly more general form.

\[ a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n) \]

(2)

The left hand side is the same as (1), but on the right hand side we have \( b^n p(n) \), where \( b \) is a constant; an\( p(n) \) is a polynomial in \( n \) of degree \( d \).

Example: Solve the following recurrence: \( t_n - 2t_{n-1} = 3^n \).

In this case \( b = 3 \) and \( p(n) = 1 \), a polynomial of degree \( d = 0 \). A little manipulation allows us to reduce this example to the form (1). To see this, we first multiply the recurrence by 3, obtaining

\[ 3t_n - 6t_{n-1} = 3^{n+1}. \]

If we replace \( n \) by \( n+1 \) in the original recurrence, we get

\[ t_{n+1} - 2t_n = 3^{n+1}. \]

Finally, subtracting these two equations, we have

\[ t_{n+1} - 5t_n + 6t_{n-1} = 0, \]

which can be solved by the above method. The characteristic equation is

\[ x^2 - 5x + 6 = 0. \]

That is, \( (x-2)(x-3) = 0 \).

Intuitively we can see that the factor \( (x-2) \) corresponds to the left-hand side of the original recurrence, whereas the factor \( (x-3) \) has appeared as a result of our manipulation to get rid of the right-hand side.
**Recurrences (cont.)**

*Generalizing this approach,* we can show that to solve (2) it is sufficient to take the following characteristic equation

\[(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x-b)^{d+1} = 0.\]

Once this equation is obtained, proceed as in the homogeneous case.

**Example:** The number of movements of a ring required in the Towers of Hanoi problem is given by the following recurrence relation:

\[t_n = 2t_{n-1} + 1, \quad n \geq 1, \text{ subject to } t_0 = 0.\]

The recurrence can be written

\[t_n - 2t_{n-1} = 1,\]

which is of the form (2) with \(b = 1, p(n) = 1,\) a polynomial of degree 0. The characteristic equation is therefore

\[(x-2)(x-1) = 0.\]

The roots of this equation are 1 and 2, so the general solution of the recurrence is

\[t_n = c_1 1^n + c_2 2^n.\]

We need two initial conditions. We know that \(t_0 = 0;\) to find a second initial condition we use the recurrence itself to calculate \(t_1 = 2t_0 + 1 = 1.\)

We finally have

\[c_1 + c_2 = 0 \quad n = 0\]
\[c_1 + 2c_2 = 1 \quad n = 1\]

From which we obtain the solution, \(t_n = 2^n - 1,\) where \(c_1 = -1, c_2 = 1.\)

Then we can conclude that \(t_n \in \Theta(2^n).\)
Recurrences (cont.)

A further generalization of the same type of argument allows us finally to solve recurrences of the form

\[ a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b_1 n p_1(n) + b_2 n p_2(n) + \ldots \] (3)

where the \( b_i \) are distinct constants and \( p_i(n) \) are polynomials in \( n \) respectively of degree \( d_i \). It suffices to write the characteristic equation

\[ (a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1} \ldots = 0, \]

**Example:** Solve the following recurrence:

\[ t_n = 2t_{n-1} + n + 2^n \quad n \geq 1, \text{ subject to } t_0 = 0. \]

\[ t_n - 2t_{n-1} = n + 2^n, \text{ which is of the form (3) with } b_1 = 1, p_1(n) = n, b_2 = 2, p_2(n) = 1. \]

The characteristic equation is \((x-2)(x-1)^2(x-2) = 0\), Roots: 1, 2, 2. The general solution of the recurrence is therefore of the form

\[ t_n = c_1 n + c_2 n^2 + c_3 2^n + c_4 n 2^n \]

Using the recurrence, we can calculate \( t_1 = 3 \) \((t_1 = 0 + 1 + 2^1 = 3)\), \( t_2 = 12\), \( t_3 = 35 \).

\[ \begin{align*}
    c_1 + c_3 &= 0 \quad n = 0, \\
    c_1 + c_2 + 2c_3 + 2c_4 &= 3 \quad n = 1 \\
    c_1 + 2c_2 + 4c_3 + 8c_4 &= 12 \quad n = 2 \\
    c_1 + 3c_2 + 8c_3 + 24c_4 &= 35 \quad n = 3, \text{ arriving at } t_n = -2 - n + 2^{n+1} + n 2^n \end{align*} \]

Therefore, \( t_n \in \Theta(n 2^n) \).
Change of Variable:

We write $T(n)$ for the term of a general recurrence, and $t_k$ for the term of a new recurrence obtained by a change of variable.

**Example:** Solve the recurrence if $n$ is a power of 2 and if

$$T(n) = 4T(n/2) + n \quad n > 1.$$ Replace $n$ by $2^k$ (so that $k = \log_2 n$) to obtain $T(2^k) = 4T(2^{k-1}) + 2^k$. This can be written as

$$t_k = 4t_{k-1} + 2^k \quad \text{if } t_k = T(2^k) = T(n).$$

We know how to solve this new recurrence: the characteristic equation is

$$(x-4)(x-2) = 0$$

and hence $t_k = c_14^k + c_22^k$.

Putting $n$ back instead of $k$, we find

$$T(n) = c_1n^2 + c_2n.$$ $T(n)$ is therefore in $O(n^2 \mid n \text{ is a power of 2}).$
Recurrences (cont.)

Example: Solve the recurrence if \( n \) is a power of 2 and if
\[
T(n) = 2T(n/2) + n \lg n \quad n > 1.
\]
Replace \( n \) by \( 2^k \) (so that \( k = \lg n \)) to obtain \( T(2^k) = 2T(2^{k-1}) + k2^k \).
This can be written as
\[
t_k = 2t_{k-1} + k2^k \quad \text{if } t_k = T(2^k) = T(n).
\]
We know how to solve this new recurrence: the characteristic equation is
\[
(x-2)^3 = 0, \quad \text{(since } t_k - 2t_{k-1} = k2^k, \text{ where } b = 2, \ p(k) = k, \text{ and } d = 1),
\]
and hence \( t_k = c_12^k + c_2k2^k + c_3k^22^k \). Putting \( n \) back instead of \( k \), we find
\[
T(n) = c_1n + c_2n\lg n + c_3n\lg^2 n.
\]
Hence \( T(n) \in O(n\lg^2 n \mid n \text{ is a power of 2}) \).

Example: Solve the recurrence if \( n \) is a power of 2 and if
\[
T(n) = 3T(n/2) + cn \quad \text{(c is constant, } n = 2^k > 1.)
\]
Replace \( n \) by \( 2^k \) (so that \( k = \lg n \)) to obtain \( T(2^k) = 3T(2^{k-1}) + c2^k \).
This can be written as
\[
t_k = 3t_{k-1} + c2^k \quad \text{if } t_k = T(2^k) = T(n).
\]
The characteristic equation is
\[
(x-3)(x-2) = 0, \quad \text{(since } t_k - 3t_{k-1} = c2^k, \text{ where } b = 2, \ p(k) = c, \text{ and } d = 0), \text{ and hence}
\]
\[
t_k = c_13^k + c_22^k \quad \text{If we put } n \text{ back, instead of } k, \text{ we find}
\]
\[
T(n) = c_13^\lg n + c_2n.
\]
Since \( a^{\log b} = b^{\log a} \), therefore, \( T(n) = c_1n^{\log 3} + c_2n \). Hence, \( T(n) \in O(n^{\log 3}) \).