Ch. 2
Analysis of Alg. Efficiency

2.1 The Analysis Framework
How to measure the size of the input

It depends on the problem! Examples:

• degree of a polynomial
• nr. of elements of an array
• the two dimensions of a matrix, either separately \((m, n)\), or as a product \(m \cdot n\)
  - If matrix is square, we can measure only \(n\)
• the maximum depth of a binary tree
• the number of bits in the binary representation of a (large) integer \(n\):  
  \[ b = \lfloor \log_2 n \rfloor + 1 \]

Verify this formula with pencil and paper for small integers!
How to measure running time

Measure the actual time (hrs, min, s, ms, etc.)

Problem: The time depends on many factors that are not algorithm-related:

- the hardware and OS of the underlying computer
- the compiler/bytecode generator/interpreter
- today’s computers usually run dozens/hundreds of processes “in parallel” (!)
How to measure running time

Identify one or more **basic operations** of the alg., i.e. the ones that take the most time. Examples:

- key comparisons in searching and sorting algs.
- divisions, multiplications and add/sub in numerical algs.
- position evaluation in chess-playing algs.

In many algs., we can isolate one basic op.

**Count** how many times the **basic operation** is executed as a function of the size of the input n.
Application

Let \( C(n) \) be the nr. of times the basic operation is executed as a function of the size of the input \( n \).

Let \( c_{op} \) be the time for the basic oper. to run on a given computer (a better notation would be \( t_{op} \)).

Then we can estimate the running time:

\[
T(n) \approx c_{op}C(n)
\]

Since we’re estimating anyway, we can retain only the most significant parts of the function \( C(n) \). Lower-order terms can be ignored:

\[
C(n) = \frac{1}{2}n(n - 1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2
\]
Application

Actually, the estimate becomes more accurate if we refer to a standard size of the input, for which the time was accurately measured:

\[ C(n) = \frac{1}{2} n(n - 1) = \frac{1}{2} n^2 - \frac{1}{2} n \approx \frac{1}{2} n^2 \]

\[
\frac{T(2n)}{T(n)} \approx \frac{c_{op} C(2n)}{c_{op} C(n)} \approx \frac{\frac{1}{2} (2n)^2}{\frac{1}{2} n^2} = 4.
\]

Under this scenario, the constant factors can also be ignored! We are left with the order of growth of the function \( C(n) \).
Order of Growth

Note: The base of the logarithm “does not matter”, because changing the base amounts to multiplying by a constant:

\[ \log_a n = \log_a b \log_b n \]
Important caveat!

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \log_2 n )</th>
<th>( n )</th>
<th>( n \log_2 n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( 2^n )</th>
<th>( n! )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.3</td>
<td>10(^1)</td>
<td>3.3(\cdot)10(^1)</td>
<td>10(^2)</td>
<td>10(^3)</td>
<td>10(^3)</td>
<td>3.6(\cdot)10(^6)</td>
</tr>
<tr>
<td>10(^2)</td>
<td>6.6</td>
<td>10(^2)</td>
<td>6.6(\cdot)10(^2)</td>
<td>10(^4)</td>
<td>10(^6)</td>
<td>1.3(\cdot)10(^30)</td>
<td>9.3(\cdot)10(^{157})</td>
</tr>
<tr>
<td>10(^3)</td>
<td>10</td>
<td>10(^3)</td>
<td>1.0(\cdot)10(^4)</td>
<td>10(^6)</td>
<td>10(^9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10(^4)</td>
<td>13</td>
<td>10(^4)</td>
<td>1.3(\cdot)10(^5)</td>
<td>10(^8)</td>
<td>10(^{12})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10(^5)</td>
<td>17</td>
<td>10(^5)</td>
<td>1.7(\cdot)10(^6)</td>
<td>10(^{10})</td>
<td>10(^{15})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10(^6)</td>
<td>20</td>
<td>10(^6)</td>
<td>2.0(\cdot)10(^7)</td>
<td>10(^{12})</td>
<td>10(^{18})</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This ordering of the functions does not hold for all values \( n \) of the input, but only \textit{from a certain value on} - see this example:
QUIZ: How does the time (nr. ops.) increase when n doubles?

**TABLE 2.1** Values (some approximate) of several functions important for analysis of algorithms

<table>
<thead>
<tr>
<th>n</th>
<th>log₂ n</th>
<th>n</th>
<th>n log₂ n</th>
<th>n²</th>
<th>n³</th>
<th>2ⁿ</th>
<th>n!</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.3</td>
<td>10¹</td>
<td>3.3·10¹</td>
<td>10²</td>
<td>10³</td>
<td>10³</td>
<td>3.6·10⁶</td>
</tr>
<tr>
<td>10²</td>
<td>6.6</td>
<td>10²</td>
<td>6.6·10²</td>
<td>10⁴</td>
<td>10⁶</td>
<td>1.3·10³⁰</td>
<td>9.3·10¹⁵⁷</td>
</tr>
<tr>
<td>10³</td>
<td>10</td>
<td>10³</td>
<td>1.0·10⁴</td>
<td>10⁶</td>
<td>10⁹</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10⁴</td>
<td>13</td>
<td>10⁴</td>
<td>1.3·10⁵</td>
<td>10⁸</td>
<td>10¹²</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10⁵</td>
<td>17</td>
<td>10⁵</td>
<td>1.7·10⁶</td>
<td>10¹⁰</td>
<td>10¹⁵</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10⁶</td>
<td>20</td>
<td>10⁶</td>
<td>2.0·10⁷</td>
<td>10¹²</td>
<td>10¹⁸</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Worst, best and average-case

Running time often depends not only on an input size but also on the specifics of a particular input.

\[
C_{\text{worst}}(n) = n \quad C_{\text{best}}(n) = 1 \quad C_{\text{ave}}(n) = \frac{(n+1)}{2}
\]
Average-case for sequential search

\( p \) is the probability to find the key in the array:

\[
C_{avg}(n) = \left[ 1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n} \right] + n \cdot (1 - p) \\
= \frac{p}{n} \left[ 1 + 2 + \cdots + i + \cdots + n \right] + n(1 - p) \\
= \frac{p}{n} \frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).
\]

What does this formula become if the key is almost always found? Almost always not found? Found about half the time?
2.1 Individual work for next time
(in notebook!)

End-of-section exercises 1, 2, 10.
2.2 Asymptotic Notations

Informal Introduction

Informally, $O(g(n))$ is the set of all functions with a lower or same order of growth as $g(n)$ (to within a constant multiple, as $n$ goes to infinity). Thus, to give a few examples, the following assertions are all true:

$$n \in O(n^2), \quad 100n + 5 \in O(n^2), \quad \frac{1}{2}n(n - 1) \in O(n^2).$$

The second notation, $\Omega(g(n))$, stands for the set of all functions with a higher or same order of growth as $g(n)$ (to within a constant multiple, as $n$ goes to infinity). For example,

$$n^3 \in \Omega(n^2), \quad \frac{1}{2}n(n - 1) \in \Omega(n^2), \quad \text{but } 100n + 5 \notin \Omega(n^2).$$

Finally, $\Theta(g(n))$ is the set of all functions that have the same order of growth as $g(n)$ (to within a constant multiple, as $n$ goes to infinity). Thus, every quadratic function $an^2 + bn + c$ with $a > 0$ is in $\Theta(n^2)$, but so are, among infinitely many others, $n^2 + \sin n$ and $n^2 + \log n$. (Can you explain why?)
2.2 Asymptotic Notations – formal definitions

\( t(n) \) is said to be in \( O(g(n)) \) \quad \text{for all } n \geq n_0

\( t(n) \) is said to be in \( \Omega(g(n)) \) \quad \text{for all } n \geq n_0

\( t(n) \) is said to be in \( \Theta(g(n)) \) \quad c_2 g(n) \leq t(n) \leq c_1 g(n) \quad \text{for all } n \geq n_0

The constants \( n_0, c, c_1, c_2 \) are for us to choose, depending on the problem. As long as we can find one combination of such constants, the proof is complete! Example on next slide.
As an example, let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$. Indeed,

$$100n + 5 \leq 100n + n \text{ (for all } n \geq 5) = 101n \leq 101n^2.$$
THEOREM  If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then
\[ t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}). \]

Proof on p.56 – read and understand!
Using limits to determine the order of growth

\[
\lim_{n \to \infty} \frac{t(n)}{g(n)} = \begin{cases} 
0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\
1 & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\
\infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). 
\end{cases} \]

---

3. The fourth case, in which such a limit does not exist, rarely happens in the actual practice of analyzing algorithms. Still, this possibility makes the limit-based approach to comparing orders of growth less general than the one based on the definitions of $O$, $\Omega$, and $\Theta$.

Although less general than the actual definitions, limits are very useful in practice because they allow us to leverage the power of calculus, in particular L’Hopital’s rule and Stirling’s formula! Examples on the next slides.
Using limits to determine the order of growth

**EXAMPLE 2**  Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$. (Unlike Example 1, the answer here is not immediately obvious.)

\[
\lim_{n \to \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \to \infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n \to \infty} \frac{(\log_2 e) \frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2 \log_2 e \lim_{n \to \infty} \frac{1}{\sqrt{n}} = 0.
\]
Using limits to determine the order of growth

**EXAMPLE 3** Compare the orders of growth of $n!$ and $2^n$. (We discussed this informally in Section 2.1.) Taking advantage of Stirling’s formula, we get

$$
\lim_{n \to \infty} \frac{n!}{2^n} = \lim_{n \to \infty} \frac{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n}{2^n} = \lim_{n \to \infty} \sqrt{2\pi n} \frac{n^n}{2^n e^n} = \lim_{n \to \infty} \sqrt{2\pi n} \left(\frac{n}{2e}\right)^n = \infty.
$$
Basic efficiency classes

Even though asymptotic notation glosses over lower-order terms and constant factors, there are still infinitely many classes of growth (in the \textit{theta} sense).

For example, in between the logarithmic and linear functions from Table 2.1, there are a lot of roots, not \textit{theta} of each other:

\[
\log n \quad \ldots \quad n^{1/5} \quad n^{1/4} \quad n^{1/3} \quad n^{1/2} \quad n^{2/3} \quad n^{3/4} \quad \ldots \quad n
\]
Nevertheless, practical algorithms usually fall within one of these basic classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td>Short of best-case efficiencies, very few reasonable examples can be given since an algorithm’s running time typically goes to infinity when its input size grows infinitely large.</td>
</tr>
<tr>
<td>log ( n )</td>
<td>logarithmic</td>
<td>Typically, a result of cutting a problem’s size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.</td>
</tr>
<tr>
<td>( n )</td>
<td>linear</td>
<td>Algorithms that scan a list of size ( n ) (e.g., sequential search) belong to this class.</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>linearithmic</td>
<td>Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>quadratic</td>
<td>Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on ( n \times n ) matrices are standard examples.</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>cubic</td>
<td>Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>exponential</td>
<td>Typical for algorithms that generate all subsets of an ( n )-element set. Often, the term “exponential” is used in a broader sense to include this and larger orders of growth as well.</td>
</tr>
<tr>
<td>( n! )</td>
<td>factorial</td>
<td>Typical for algorithms that generate all permutations of an ( n )-element set.</td>
</tr>
</tbody>
</table>
2.2 Individual work for next time (in notebook!)

End-of-section exercise 2
QUIZ – Order of Growth

Consider the functions $\sqrt{n}$ and $n^{\sin(n)}$. Decide and then prove if either one is Big Oh, Big Omega or Theta of each other.

• Hint: Start by comparing $\sqrt{n}$ and $n$.
• Hint: Sketch the functions! Use, for example https://www.desmos.com/calculator
Consider the functions $\sqrt{n}$ and $n^{\sin(n)}$. Decide and then prove if either one is Big Oh, Big Omega or Theta of each other.

**Answer:**

$\sqrt{n}$ is $O(n)$, $n$ is $\Omega(\sqrt{n})$, neither is $\Theta$ of the other

$\sqrt{n}$ and $n^{\sin(n)}$ are neither $O$, $\Omega$ or $\Theta$ of each other!
2.3 Analysis of Non-recursive algs.

```
ALGORITHM MaxElement(A[0..n – 1])
  //Determines the value of the largest element in a given array
  //Input: An array A[0..n – 1] of real numbers
  //Output: The value of the largest element in A
  maxval ← A[0]
  for i ← 1 to n – 1 do
    if A[i] > maxval
      maxval ← A[i]
  return maxval
```

a. How to measure the input size?
b. What are the basic operations (that get executed the most)?
c. Should we consider them both, or are there grounds for preferring one over the other?
d. Best, worst, average cases?
ALGORITHM\hspace{1em}MaxElement(A[0..n − 1])

// Determines the value of the largest element in a given array
// Input: An array A[0..n − 1] of real numbers
// Output: The value of the largest element in A

maxval \leftarrow A[0]

\text{for } i \leftarrow 1 \text{ to } n − 1 \text{ do}

\begin{align*}
\quad & \text{if } A[i] > maxval \\
\quad & \hspace{1em} maxval \leftarrow A[i]
\end{align*}

\text{return } maxval

\[
C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)
\]
The plan (algorithm? 😊)

1. Decide on a parameter (or parameters) indicating an input’s size.
2. Identify the algorithm’s basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm’s basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

\[ \sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n + 1)}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2) \]

See App.A for other useful formulas!
ALGORITHM $UniqueElements(A[0..n - 1])$
// Determines whether all the elements in a given array are distinct
// Input: An array $A[0..n - 1]$
// Output: Returns “true” if all the elements in $A$ are distinct
// and “false” otherwise
for $i \leftarrow 0$ to $n - 2$ do
  for $j \leftarrow i + 1$ to $n - 1$ do
return true

Extra-credit question ...
**ALGORITHM**  \textit{UniqueElements}(A[0..n - 1])

// Determines whether all the elements in a given array are distinct
// Input: An array A[0..n - 1]
// Output: Returns “true” if all the elements in A are distinct
// and “false” otherwise

\[\text{for } i \leftarrow 0 \text{ to } n - 2 \text{ do} \]
\[\text{for } j \leftarrow i + 1 \text{ to } n - 1 \text{ do} \]
\[\text{if } A[i] = A[j]\text{ return false} \]
\[\text{return true} \]

a. How to measure the input size? \(n\)
b. What is the basic operation (that gets executed the most)? \textbf{Comparison}
c. Best, worst, average cases? See next slide.
Worst case: Either no early return, or return triggered for the last pair of elements examined.

\[
C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n - 1) - (i + 1) + 1] = \sum_{i=0}^{n-2} (n - 1 - i)
\]

\[
= \sum_{i=0}^{n-2} (n - 1) - \sum_{i=0}^{n-2} i = (n - 1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}
\]

\[
= (n - 1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).
\]
\[
\text{row } i \begin{bmatrix}
\vdots \\
\vdots \\
\end{bmatrix} \times \begin{bmatrix}
\vdots \\
\vdots \\
\end{bmatrix} = \begin{bmatrix}
C[i,j]
\end{bmatrix}
\]

col. j

**ALGORITHM**  \text{MatrixMultiplication}(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])

//Multiplies two square matrices of order \( n \) by the definition-based algorithm
//Input: Two \( n \times n \) matrices \( A \) and \( B \)
//Output: Matrix \( C = AB \)

\textbf{for} \( i \leftarrow 0 \textbf{ to } n - 1 \) \textbf{ do}

\hspace{1em} \textbf{for} \( j \leftarrow 0 \textbf{ to } n - 1 \) \textbf{ do}

\hspace{2em} \( C[i, j] \leftarrow 0.0 \)

\hspace{2em} \textbf{for} \( k \leftarrow 0 \textbf{ to } n - 1 \) \textbf{ do}

\hspace{3em} \( C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j] \)

\textbf{return} \( C \)
a. How to measure the input size?
b. What is the basic operation (that gets executed the most)?
c. Best, worst, average cases?

a. How to measure the input size? \( n \)
b. What is the basic operation (that gets executed the most)? \textbf{Multiplication}
c. Best, worst, average cases? \textbf{They are all the same!}
Read and take notes:

Example 4/p.66:
Finding number of bits to represent an integer.
2.3 Individual work for next time (in notebook!)

End-of-section exercises 1, 4.
ALGORITHM \( \text{MatrixMultiplication}(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1]) \)

//Multiplies two square matrices of order \( n \) by the definition-based algorithm
//Input: Two \( n \times n \) matrices \( A \) and \( B \)
//Output: Matrix \( C = AB \)
for \( i \leftarrow 0 \) to \( n - 1 \) do 
  for \( j \leftarrow 0 \) to \( n - 1 \) do 
    \( C[i, j] \leftarrow 0.0 \)
    for \( k \leftarrow 0 \) to \( n - 1 \) do
      \( C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j] \)
return \( C \)

\[
M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3
\]

If it is also important to account for additions:

\[
T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3
\]
2.4 Analysis of Recursive algs.

**EXAMPLE 1** Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer $n$. Since

$$n! = 1 \cdot \ldots \cdot (n - 1) \cdot n = (n - 1)! \cdot n \quad \text{for } n \geq 1$$

and $0! = 1$ by definition, we can compute $F(n) = F(n - 1) \cdot n$ with the following recursive algorithm.

**ALGORITHM** $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer $n$
//Output: The value of $n!$
if $n = 0$ return $1$
else return $F(n - 1) \cdot n$

a. How to measure the input size?
b. What is the basic operation (that gets executed the most)?
c. Best, worst, average cases?
Idea: Since the algorithm itself is recursive, let us express its efficiency recursively as well!

Recurrence relation

\[ M(n) = M(n - 1) + 1 \quad \text{for} \quad n > 0, \]
\[ M(0) = 0. \]

What is missing here?

We solve by back-substitution:

\[ M(n) = M(n - 1) + 1 = \cdots = M(n - i) + i = \cdots = M(n - n) + n = n. \]
1. Solve the following recurrence relations.
   a. $x(n) = x(n - 1) + 5$ for $n > 1$, \hspace{10pt} x(1) = 0$
   b. $x(n) = 3x(n - 1)$ for $n > 1$, \hspace{10pt} x(1) = 4$
The plan

General Plan for Analyzing the Time Efficiency of Recursive Algorithms
1. Decide on a parameter (or parameters) indicating an input’s size.
2. Identify the algorithm’s basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.
Individual work for next time (in notebook!)

End-of-section exercise 1 c, d, e. Use back-substitution!
2.4 Analysis of Recursive algs. - ToH

a. How to measure the input size?
b. What is the basic operation (that gets executed the most)?
c. Best, worst, average cases?

We’re only examining the best case (optimal solution) this time, since the input changes only with n, the # of disks.
ToH – Recurrence relation

Figure 2.4 Recursive solution to the Tower of Hanoi puzzle.

\[ M(n) = 2M(n-1) + 1 \quad \text{for } n > 1, \quad M(1) = 1. \]

Solve by backward substitution:

\[
M(n) = 2M(n-1) + 1 \\
= 2[2M(n-2) + 1] + 1 = 2^2M(n-2) + 2 + 1 \quad \text{sub. } M(n-2) = 2M(n-3) + 1 \\
= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3M(n-3) + 2^2 + 2 + 1. 
\]

The pattern of the first three sums on the left suggests that the next one will be 
\[ 2^4M(n-4) + 2^3 + 2^2 + 2 + 1, \] and generally, after \( i \) substitutions, we get 
\[ M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1. \]

Since the initial condition is specified for \( n = 1 \), which is achieved for \( i = n - 1 \), we get the following formula for the solution to recurrence (2.3):

\[
M(n) = 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1 \\
= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.
\]
ToH – Visualizing the tree of recursive calls
2.5 Computing $n^{\text{th}}$ Fibonacci Number

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

\[ F(n) = F(n - 1) + F(n - 2) \quad \text{for } n > 1 \]
\[ F(0) = 0, \quad F(1) = 1. \]

Back-substitution doesn’t work directly!

We are going to deploy a powerful “machine” for solving recurrence relations:

Formula (Theorem) for **Linear Recurrences with Constant Coefficients**
Solving a **Linear Recurrence w/Constant Coefficients**

\[
F(n) = F(n - 1) + F(n - 2) \quad \text{for } n > 1 \quad \quad F(0) = 0, \quad F(1) = 1.
\]

\[
ax(n) + bx(n - 1) + cx(n - 2) = 0.
\]

The Theorem (See App.B) says that the solution to the recurrence is a linear combination of powers of the solutions of the associated **characteristic equation**:

\[
ax^2 + bx + c = 0
\]

\[
x^2 - x - 1 = 0
\]
\[ F(n) = F(n - 1) + F(n - 2) \quad \text{for } n > 1 \quad \text{with } F(0) = 0, \quad F(1) = 1. \]

\[ ax(n) + bx(n - 1) + cx(n - 2) = 0. \]

Solve quadratic characteristic equation

\[ r_1 = \frac{1 + \sqrt{5}}{2} \quad \text{Golden Ratio} \approx 1.618 \]

\[ r_2 = \frac{1 - \sqrt{5}}{2} \quad \approx -0.618 \]

The recurrence solution must be of the form

\[ F_n = pr_1^n + qr_2^n \]

Find the coefficients \( p \) and \( q \) using the boundary conditions:

\[ F_n = \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left( \frac{1 - \sqrt{5}}{2} \right)^n \quad \text{Binet’s formula} \]
Since the second root is <1 in magnitude, the second term tends to 0 when $n \to \infty$, so we have

$$F(n) = \frac{1}{\sqrt{5}} \phi^n$$ rounded to the nearest integer.

What is the efficiency/order of growth of this algorithm?
2.5 Computing $n^{th}$ Fibonacci Number

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$$F(n) = F(n - 1) + F(n - 2) \quad \text{for } n > 1$$

$$F(0) = 0, \quad F(1) = 1.$$  

A second algorithm:

```
ALGORITHM $F(n)$
//Computes the $n$th Fibonacci number recursively by using its definition
//Input: A nonnegative integer $n$
//Output: The $n$th Fibonacci number
if $n \leq 1$ return $n$
else return $F(n - 1) + F(n - 2)$
```
ALGORITHM  \( F(n) \)

// Computes the \( n \)th Fibonacci number recursively by using its definition
// Input: A nonnegative integer \( n \)
// Output: The \( n \)th Fibonacci number
\[
\text{if } n \leq 1 \text{ return } n \\
\text{else return } F(n - 1) + F(n - 2)
\]

\[
A(n) = A(n - 1) + A(n - 2) + 1 \quad \text{for } n > 1, \\
A(0) = 0, \quad A(1) = 0.
\]

Note that this is an \textbf{inhomogeneous recurrence}!

We can reduce our inhomogeneous recurrence to a homogeneous one by rewriting it as

\[
[A(n) + 1] - [A(n - 1) + 1] - [A(n - 2) + 1] = 0
\]

and substituting \( B(n) = A(n) + 1 \):

\[
B(n) - B(n - 1) - B(n - 2) = 0, \\
B(0) = 1, \quad B(1) = 1.
\]
ALGORITHM $F(n)$

//Computes the $n$th Fibonacci number recursively by using its definition
//Input: A nonnegative integer $n$
//Output: The $n$th Fibonacci number
if $n \leq 1$ return $n$
else return $F(n - 1) + F(n - 2)$

$$B(n) - B(n - 1) - B(n - 2) = 0,$$
$$B(0) = 1, \quad B(1) = 1.$$ 

This is simply the Fibonacci sequence shifted by one!

$$A(n) = B(n) - 1 = F(n + 1) - 1 = \frac{1}{\sqrt{5}}(\phi^{n+1} - \phi^{n+1}) - 1.$$ 

$$A(n) \in \Theta(\phi^n)$$
Algorithm using the non-recursive (iterative) definition:

\[ A(n) = n - 1 \in \Theta(n) \]
2.6 Empirical Analysis of Algs.

We use it when the mathematical analysis is too difficult (or inexistent!)

**General Plan for the Empirical Analysis of Algorithm Time Efficiency**

1. Understand the experiment’s purpose.
2. Decide on the efficiency metric $M$ to be measured and the measurement unit (an operation count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.
5. Generate a sample of inputs.
6. Run the algorithm (or algorithms) on the sample’s inputs and record the data observed.
7. Analyze the data obtained.
Assigned reading for next time (take notes in notebook!)

Read the entire Sec. 2.6.

Think about what we have been doing in the lab with the “gloves” puzzle!
Homework for Ch.2
- due next Monday, Sept. 16 -

- Ex. 2.1 (p.21): 7, 9abc (only a, b, and c)
  - Note: For prob.9, use the limit technique from p.57 of our text!

- Ex. 2.2 (p.60): 5, 6ab
  - Note: No proofs are required for prob.5, simply write down the list of functions. Prob. 6 requires proofs!

- Ex. 2.3 (p.67): 2, 5

- Ex. 2.4 (p.76): 3, 4

- Ex. 2.5 (p.83): 2, 6