An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.
Math you need to Review

- Series summations
- Logarithms and Exponents
- Proof techniques
- Basic probability

**properties of logarithms:**
\[ \log_b(xy) = \log_b x + \log_b y \]
\[ \log_b (x/y) = \log_b x - \log_b y \]
\[ \log_b x^a = a \log_b x \]
\[ \log_b a = \log_x a / \log_x b \]

**properties of exponentials:**
\[ a^{(b+c)} = a^b a^c \]
\[ a^{bc} = (a^b)^c \]
\[ a^b / a^c = a^{(b-c)} \]
\[ b = a^{\log_a b} \]
\[ b^c = a^{c \log_a b} \]
Sum series-examples

\[
\sum_{i=0}^{\infty} \left( \frac{1}{2} \right)^i = \lim_{n \to \infty} \sum_{i=0}^{n} \left( \frac{1}{2} \right)^i
\]

\[
= \lim_{n \to \infty} \left( 2 - \left( \frac{1}{2} \right)^n \right)
\]

\[= 2.\]
\[
\sum_{i=-\infty}^{n} 2^i = \left( \sum_{i=-\infty}^{0} 2^i \right) + \left( \sum_{i=0}^{n} 2^i \right) - 1 \\
= \left( \sum_{i=0}^{\infty} \left( \frac{1}{2} \right)^i \right) + \left( \sum_{i=0}^{n} 2^i \right) - 1 \\
= (2) + (2^{n+1} - 1) - 1 \\
= 2^{n+1}
\]
\[ \begin{align*}
S_n &= \sum_{i=0}^{n} \frac{i}{2^i} \\
2S_n &= 2 \sum_{i=0}^{n} \frac{i}{2^i} \\
&= \sum_{i=0}^{n} \frac{i}{2^{i-1}} \\
&= \sum_{j=-1}^{n-1} \frac{j + 1}{2^j} \\
&= \sum_{j=0}^{n} \frac{j + 1}{2^j} - \frac{n + 1}{2^n}
\end{align*} \]

\[ \begin{align*}
2S_n - S_n &= \sum_{j=0}^{n} \frac{1}{2^j} - \frac{n + 1}{2^n} \\
S_n &= (2 - (1/2)^n) - (n + 1)/2^n \\
\sum_{i=0}^{\infty} \frac{i}{2^i} &= \lim_{n \to \infty} \sum_{i=0}^{n} \frac{i}{2^i} \\
&= \lim_{n \to \infty} S_n \\
&= \lim_{n \to \infty} (2 - (1/2)^n) - (n + 1)/2^n \\
&= 2.
\]
Recurrence Relations

\[ T(n) = T(n - 1) + 1, \quad n > 0 \]
\[ = (T(n - 2) + 1) + 1 \]
\[ = ((T(n - 3) + 1) + 1) + 1 \]
\[ \vdots \]
\[ = T(n - k) + k \]
\[ \vdots \]
\[ = T(0) + n, \quad n - k = 0 \]
\[ = n + 1. \]
Assume that \( n = ma \):

\[
T(n) = T(n - a) + 1, \quad n > a
\]

\[
T(ma) = T(((m - 1)a) + 1
\]

\[
= (T(((m - 2)a) + 1) + 1
\]

\[
= ((T(((m - 3)a) + 1) + 1) + 1
\]

\[
\vdots
\]

\[
= T(((m - k)a) + k
\]

\[
\vdots
\]

\[
= T(a) + (m - 1), \quad (m - k) = 1
\]

\[
= m
\]

\[
= n/a.
\]
\[
T(n) = 2T(n - 1) + 1, \quad n > 0 \\
= 2(2T(n - 2) + 1) + 1 \\
= 2(2(2T(n - 3) + 1) + 1) + 1 \\
\vdots \\
= 2^k T(n - k) + \sum_{i=0}^{k-1} 2^i \\
\vdots \\
= 2^n T(0) + \sum_{i=0}^{n-1} 2^i, \quad (n - k) = 0 \\
= 2^n + 2^n - 1 \\
= 2^{n+1} - 1.
\]
\[ T(n) = \begin{cases} T(n/2) + 1, & n > 1 \\ (T(n/4) + 1) + 1 \\ ((T(n/8) + 1) + 1) + 1 \\ \vdots \\ T(n/2^k) + k \\ \vdots \\ T(1) + \log_2 n, & (n/2^k) = 1 \\ 1 + \log_2 n. \end{cases} \]
\[ T(n) = \begin{align*}
&= 2T(n/2) + 1, \quad n > 1 \\
&= 2(2T(n/4) + 1) + 1 \\
&= 2(2(2T(n/8) + 1) + 1) + 1 \\
&\vdots \\
&= 2^k T(n/2^k) + \sum_{i=0}^{k-1} 2^i \\
&\vdots \\
&= nT(1) + n - 1, \quad (n/2^k) = 1 \\
&= 2n - 1.
\]
\[ T(n) = 2T\left(\frac{n}{2}\right) + n, \quad n > 1 \]
\[ = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n \]
\[ = 2\left(2\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + \frac{n}{2}\right) + n \]
\[ \vdots \]
\[ = 2^k T\left(\frac{n}{2^k}\right) + kn \]
\[ \vdots \]
\[ = nT(1) + n \log_2 n, \quad \left(\frac{n}{2^k}\right) = 1 \]
\[ = n + n \log_2 n. \]
Assumptions for the computational model

- Basic computer with sequentially executed instructions
- Infinite memory
- Has standard operations; addition, multiplication, comparison in 1 time unit unless stated otherwise
- Has fixed-size (32bits) integers such that no-fancy operations are allowed!!! eg. matrix inversion,
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance, image, robotics, AI, etc.
Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Assume a subprogram is written to get an accurate measure of the actual running time
- Plot the results

![Graph showing time (ms) vs. input size]
Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used.
Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation.
- Characterizes running time as a function of the input size, $n$.
- Takes into account all possible inputs.
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment.
Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm $arrayMax(A, n)$

Input array $A$ of $n$ integers

Output maximum element of $A$

$currentMax \leftarrow A[0]$

for $i \leftarrow 1$ to $n - 1$ do

  if $A[i] > currentMax$ then
    $currentMax \leftarrow A[i]$

return $currentMax$
Pseudocode Details

- **Control flow**
  - if ... then ...
  - while ... do ...
  - repeat ... until ...
  - for ... do ...
  - Indentation replaces braces

- **Method declaration**
  Algorithm `method (arg [, arg ...])`
  
  Input ...
  Output ...

- **Method call**
  `var.method (arg [, arg ...])`

- **Return value**
  `return expression`

- **Expressions**
  - Assignment (like `=` in Java)
  - Equality testing (like `==` in Java)
  - $n^2$ Superscripts and other mathematical formatting allowed
Important Functions

Seven functions that often appear in algorithm analysis:

- Constant \( \approx 1 \)
- Logarithmic \( \approx \log n \)
- Linear \( \approx n \)
- N-Log-N \( \approx n \log n \)
- Quadratic \( \approx n^2 \)
- Cubic \( \approx n^3 \)
- Exponential \( \approx 2^n \)

In a log-log chart, the slope of the line corresponds to the growth rate of the function.
Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important
- Assumed to take a constant amount of time in the RAM model

Examples:
- Evaluating an expression
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method
Detailed Model 4 counting operations-1

The time required for the following operations are all constants.

<table>
<thead>
<tr>
<th>$\tau_{\text{fetch}}$</th>
<th>to fetch an integer operand from memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_{\text{store}}$</td>
<td>to store an integer result in memory</td>
</tr>
<tr>
<td>$\tau_{+}$</td>
<td>to add two integers</td>
</tr>
<tr>
<td>$\tau_{-}$</td>
<td>to subtract two integers</td>
</tr>
<tr>
<td>$\tau_{x}$</td>
<td>to multiply two integers</td>
</tr>
<tr>
<td>$\tau_{/}$</td>
<td>to divide two integers</td>
</tr>
<tr>
<td>$\tau_{&lt;}$</td>
<td>to comparison of two integers</td>
</tr>
</tbody>
</table>
Detailed Model 4 counting operations-2

The time require for the following operations are all constants.

<table>
<thead>
<tr>
<th>( \tau_{\text{return}} )</th>
<th>to return from a method</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_{\text{call}} )</td>
<td>to call a method</td>
</tr>
<tr>
<td>( \tau_{\text{store}} )</td>
<td>to pass an integer argument to a method</td>
</tr>
<tr>
<td>( \tau_{[.]} )</td>
<td>for the address calculation (not including the computation of the subscription)</td>
</tr>
<tr>
<td>( \tau_{\text{new}} )</td>
<td>to allocate a fixed amount of storage from the heap using new</td>
</tr>
<tr>
<td>Statement</td>
<td>Time required</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>y = x;</td>
<td>$\tau_{\text{fetch}} + \tau_{\text{store}}$</td>
</tr>
<tr>
<td>y = 1;</td>
<td>$\tau_{\text{fetch}} + \tau_{\text{store}}$</td>
</tr>
<tr>
<td>y = y + 1;</td>
<td>$2 \tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}$</td>
</tr>
<tr>
<td>y += 1;</td>
<td>$2 \tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}$</td>
</tr>
<tr>
<td>++y;</td>
<td>$2 \tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}$</td>
</tr>
<tr>
<td>y++;</td>
<td>$2 \tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}$</td>
</tr>
</tbody>
</table>
Detailed Model...  Example 1: Sum

\[ \sum_{i=1}^{n} i \]

... 

```java
public static int sum(int n) {
    int result = 0;
    for (int i = 1; i <= n; ++i){
        result += i;
    }
    return result;
}
```

<table>
<thead>
<tr>
<th>Statement</th>
<th>Time required</th>
</tr>
</thead>
<tbody>
<tr>
<td>int result = 0;</td>
<td>(\tau_{\text{fetch}} + \tau_{\text{store}})</td>
</tr>
<tr>
<td>int i = 1</td>
<td>(\tau_{\text{fetch}} + \tau_{\text{store}})</td>
</tr>
<tr>
<td>i &lt;= n</td>
<td>((2\tau_{\text{fetch}} + \tau_{&lt;})(n+1))</td>
</tr>
<tr>
<td>++i</td>
<td>((2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}})n)</td>
</tr>
<tr>
<td>result += i</td>
<td>((2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}})n)</td>
</tr>
<tr>
<td>return result</td>
<td>(\tau_{\text{fetch}} + \tau_{\text{return}})</td>
</tr>
</tbody>
</table>
Detailed Model... Example2 (*)

\[ y = a[i] \]

\[ 3\tau_{\text{fetch}} + \tau_{\text{[.]} } + \tau_{\text{store}} \]

- fetch a (the base address of the array)
- fetch i (the index into the array)
- address calculation
- fetch array element \( a[i] \)
- store the result
Detailed Model...Example 3: Horner (*)

\[
\sum_{i=0}^{n} a_i x^i
\]

public class Horner {

    public static void main(String[] args) {
        Horner h = new Horner();
        int[] a = {1, 3, 5};
        System.out.println("a(1)=" + h.horner(a, a.length - 1, 1));
        System.out.println("a(2)=" + h.horner(a, a.length - 1, 2));
    }

    int horner(int[] a, int n, int x) {
        int result = a[n];
        for (int i = n - 1; i >= 0; --i) {
            result = result * x + a[i];
            /**/System.out.println("i=" + i + " result=" + result);
        }
        return result;
    }
}

Output:

i=1 result=8
i=0 result=9
i=1 result=13
i=0 result=27
a(2)=27
public class FindMaximum {

    public static void main(String[] args) {
        FindMaximum h = new FindMaximum();
        int[] a = {1, 3, 5};
        System.out.println("max=" + h.findMaximum(a));
    }

    int findMaximum(int[] a) {
        int result = a[0];
        for (int i = 0; i < a.length; ++i) {
            if (result < a[i]) {
                result = a[i];
            }
        }
        System.out.println("i=" + i + " result=" + result);
        return result;
    }
}

3-fetch + τ[.] + τ-store
fetch + τ-store
(2-fetch + τ<) n
(2-fetch + τ+ + τ-store) (n-1)
(4-fetch + τ[.] + τ<) (n-1)
(3-fetch + τ[.] + τ-store) ?
fetch + τ-store

output

i=0 result=1
i=1 result=3
i=2 result=5
max=5
Simplified Model ... More Simplification

- All timing parameters are expressed in units of clock cycles. In effect, $T=1$.
- The proportionality constant, $k$, for all timing parameters is assumed to be the same: $k=1$. 
public class FindMaximum {

    public static void main(String[] args) {
        FindMaximum h = new FindMaximum();
        int[] a = {1, 3, 5};
        System.out.println("max=" + h.findMaximum(a));
    }

    int findMaximum(int[] a) {
        int result = a[0];
        for (int i = 0; i < a.length; ++i) {
            if (result < a[i]) {
                result = a[i];
            }
            System.out.println("i=" + i + " result=" + result);
        }
        return result;
    }
}

1 2 3 4 5 6 7 8 9 10

i=0 result=1
i=1 result=3
i=2 result=5
max=5

out

detailed
2 \(3 \tau \text{fetch} + \tau[.] + \tau \text{store}\)
3a \(\tau \text{fetch} + \tau \text{store}\)
3b \((2 \tau \text{fetch} + \tau <) n\)
3c \((2 \tau \text{fetch} + \tau + + \tau \text{store}) (n-1)\)
4 \((4 \tau \text{fetch} + \tau[.] + \tau <) (n-1)\)
6 \((3 \tau \text{fetch} + \tau[.] + \tau \text{store})?\)
9 \(\tau \text{fetch} + \tau \text{store}\)

simple
2 5
3a 2
3b \((3)n\)
3c \((4)(n-1)\)
4 \((6)(n-1)\)
6 \((5)?\)
9 2
public class GeometrikSeriesSum {

    public static void main(String[] args) {
        System.out.println("1, 4: " + geometricSeriesSum(1, 4));
        System.out.println("2, 4: " + geometricSeriesSum(2, 4));
    }

    public static int geometricSeriesSum(int x, int n) {
        int sum = 0;
        for (int i = 0; i <= n; ++i) {
            int prod = 1;
            for (int j = 0; j < i; ++j) {
                prod *= x;
            }
            sum += prod;
        }
        return sum;
    }

    x=1, n=4: a4=5
    x=2, n=4: a4=31

    output

    simple
    1
    2 2
    3a 2
    3b 3(n+2)
    3c 4(n+1)
    4 2(n+1)
    5a 2(n+1)
    5b 3 \sum_{i=0}^{n} (i+1)
    5c 4 \sum_{i=0}^{n} i
    6 4 \sum_{i=0}^{n} i^2
    7
    8 4(n+1)
    9
    10 2
    11
    12

    Total 11/2 n^2 + 47/2 n +27
Simplified Model > Algorithm_SumHorner

Geometric Series Sum ...

```java
public class GeometrikSeriesSumHorner {

    public static void main(String[] args) {
        System.out.println("1, 4: " + geometricSeriesSum(1, 4));
        System.out.println("2, 4: " + geometricSeriesSum(2, 4));
    }

    public static int geometricSeriesSum(int x, int n) {
        int sum = 0;
        for (int i = 0; i <= n; ++i) {
            sum = sum * x + 1;
        }
        return sum;
    }
}
```

Observe:
Let sum = \( a_i \),
a0 = 0
a1 = 1, a2 = x + 1, a3 = x^2 + x + 1, \ldots an = x^n + x^{n-1} + x^{n-2} + \ldots + 1
x = 1, n = 4, output: 1 + 1 + 1 + 1 + 1 = 5
x = 2, n = 4, output: 2^0 + 2^1 + 2^2 + 2^3 + 2^4 = 31

\[
\sum_{i=0}^{n} x^i
\]
```java
public class GeometrikSeriesSumPower {

    public static void main(String[] args) {
        System.out.println("1, 4: "+ powerA(1, 4));
        System.out.println("1, 4: "+ powerB(1, 4));
        System.out.println("2, 4: "+ powerA(2, 4));
        System.out.println("2, 4: "+ powerB(2, 4));
    }

    public static int powerA(int x, int n) {
        int result = 1;
        for (int i = 1; i <= n; ++i) {
            result *= x;
        }
        return result;
    }

    public static int powerB(int x, int n) {
        if (n == 0) {
            return 1;
        } else if (n % 2 == 0) { // n is even
            return powerB(x * x, n / 2);
        } else { // n is odd
            return x * powerB(x * x, n / 2);
        }
    }
}
```

```
powerB

\[ x^n = \begin{cases} 
1 & \text{n=0} \\
(x^2)^{\lfloor n/2 \rfloor} & \text{0<n, n is even} \\
x(x^2)^{\lfloor n/2 \rfloor} & \text{0<n, n is odd}
\end{cases} \]
```

```

<table>
<thead>
<tr>
<th>n</th>
<th>n=0</th>
<th>n even</th>
<th>n odd</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>10+T(\lfloor n/2 \rfloor)</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>12+T(\lfloor n/2 \rfloor)</td>
</tr>
<tr>
<td>Total</td>
<td>5</td>
<td>18+T(\lfloor n/2 \rfloor)</td>
<td>20+T(\lfloor n/2 \rfloor)</td>
</tr>
</tbody>
</table>
```
Let $n = 2^k$ for some $k > 0$.

Since $n$ is even, $\lfloor n/2 \rfloor = n/2 = 2^{k-1}$.

For $n = 2^k$, $T(2^k) = 18 + T(2^{k-1})$.

Using repeated substitution

$T(2^k) = 18 + T(2^{k-1})$
$\quad = 18 + 18 + T(2^{k-2})$
$\quad = 18 + 18 + 18 + T(2^{k-3})$
$\quad \ldots$
$\quad = 18j + T(2^{k-j})$

Substitution stops when $k = j$

$T(2^k) = 18k + T(1)$
$\quad = 18k + 20 + T(0)$
$\quad = 18k + 20 + 5$
$\quad = 18k + 25.$

$n = 2^k$ then $k = \log_2 n$

$T(2^k) = 18 \log_2 n + 25$

\[ x^n =\begin{cases} 18 + T(\lfloor n/2 \rfloor) & 0 < n \end{cases} \]
Suppose $n = 2^k - 1$ for some $k > 0$.

Since $n$ is odd, $\lfloor n/2 \rfloor = \lfloor (2^k - 1)/2 \rfloor = (2^k - 2)/2 = 2^{k-1}$

For $n = 2^k - 1$,
$T(2^k - 1) = 20 + T(2^{k-1} - 1), k > 1$.

Using repeated substitution

$T(2^k - 1) = 20 + T(2^{k-1} - 1)$
$= 20 + 20 + T(2^{k-2} - 1)$
$= 20 + 20 + 20 + T(2^{k-3} - 1)$
$\cdots$
$= 20j + T(2^{k-j} - 1)$

Substitution stops when $k = j$

$T(2^k - 1) = 20k + T(2^0 - 1)$
$= 20k + T(0)$
$= 20k + 5$.

$n = 2^k - 1$ then $k = \log_2 (n+1)$

$T(n) = 20 \log_2 (n+1) + 5$

Therefore, powerB

$x^n = \begin{cases} 
5 & n=0 \\
18 + T(\lfloor n/2 \rfloor) & 0 < n, n \text{ is even} \\
20 + T(\lfloor n/2 \rfloor) & 0 < n, n \text{ is odd} 
\end{cases}$

Average: $19(\lfloor \log_2(n+1) \rfloor + 1) + 18$
public class GeometrikSeriesSumPower {

    public static void main(String[] args) {
        System.out.println("s 2, 4: " + geometrikSeriesSumPower(2, 4));
    }

    ... 

    public static int geometrikSeriesSumPower(int x, int n) {
        return powerB(x, n + 1) - 1 / (x - 1);
    }

    public static int powerB(int x, int n) {
        if (n == 0) {
            return 1;
        } else if (n % 2 == 0) { // n is even
            return powerB(x * x, n / 2);
        } else { // n is odd
            return x * powerB(x * x, n / 2);
        }
    }
}

x=2, n=4, a4=31
X must be power of 2
Comparison of 3 Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>T(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum</td>
<td>$\frac{11}{2} n^2 + \frac{47}{2} n + 27$</td>
</tr>
<tr>
<td>Horner</td>
<td>$13n + 22$</td>
</tr>
<tr>
<td>Power</td>
<td>$19(\lceil \log_2(n+1) \rceil + 1) + 18$</td>
</tr>
</tbody>
</table>
Counting Primitive Operations for Pseudocodes

By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

Algorithm \( \text{arrayMax}(A, n) \)

\[
\begin{align*}
\text{currentMax} & \leftarrow A[0] \\
\text{for } i & \leftarrow 1 \text{ to } n - 1 \text{ do} \\
& \quad \text{if } A[i] > \text{currentMax} \text{ then} \\
& \quad \quad \text{currentMax} \leftarrow A[i] \\
& \quad \{ \text{ increment counter } i \} \\
\text{return } \text{currentMax}
\end{align*}
\]

\[
\begin{array}{|l|}
\hline
\text{# operations} & 2 & 2n & 2(n - 1) & 2(n - 1) & 2(n - 1) & 1 & 8n - 2 \\
\hline
\end{array}
\]
Estimating Running Time

Algorithm \textit{arrayMax} executes $8n - 2$ primitive operations in the worst case. Define:

- $a = \text{Time taken by the fastest primitive operation}$
- $b = \text{Time taken by the slowest primitive operation}$

Let $T(n)$ be worst-case time of \textit{arrayMax}. Then

$$a(8n - 2) \leq T(n) \leq b(8n - 2)$$

Hence, the running time $T(n)$ is bounded by two linear functions
Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation.

To perform the asymptotic analysis:
- We find the worst-case number of primitive operations executed as a function of the input size.
- We express this function with big-Oh notation.

Example:
- We determine that algorithm $arrayMax$ executes at most $8n - 2$ primitive operations.
- We say that algorithm $arrayMax$ “runs in $O(n)$ time.”

Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations.
Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$

- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm $arrayMax$
Constant Factors

The growth rate is not affected by
- constant factors or
- lower-order terms

Examples
- $10^2n + 10^5$ is a linear function
- $10^5n^2 + 10^8n$ is a quadratic function
Big-Oh Notation

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$.

Example: $2n + 10$ is $O(n)$
- $2n + 10 \leq cn$
- $(c - 2) n \geq 10$
- $n \geq 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$
Big-Oh Example

Example: the function \( n^2 \) is not \( O(n) \)
- \( n^2 \leq cn \)
- \( n \leq c \)
- The above inequality cannot be satisfied since \( c \) must be a constant
More Big-Oh Examples

- **$7n - 2$**
  - $7n - 2$ is $O(n)$
  - need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq c \cdot n$ for $n \geq n_0$
  - this is true for $c = 7$ and $n_0 = 1$

- **$3n^3 + 20n^2 + 5$**
  - $3n^3 + 20n^2 + 5$ is $O(n^3)$
  - need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$
  - this is true for $c = 4$ and $n_0 = 21$

- **$3 \log n + 5$**
  - $3 \log n + 5$ is $O(\log n)$
  - need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$
  - this is true for $c = 8$ and $n_0 = 2$
Big-Oh and Growth Rate

The big-Oh notation gives an upper bound on the growth rate of a function.

The statement “$f(n)$ is $O(g(n))$” means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.

We can use the big-Oh notation to rank functions according to their growth rate:

<table>
<thead>
<tr>
<th></th>
<th>$f(n)$ is $O(g(n))$</th>
<th>$g(n)$ is $O(f(n))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g(n)$ grows more</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$f(n)$ grows more</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Same growth</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
More Example

<table>
<thead>
<tr>
<th>$f(n)$</th>
<th>$g(n)$</th>
<th>$f(n) = O(g(n))$</th>
<th>$g(n) = O(f(n))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10n$</td>
<td>$n^2 - 10n$</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>$n^3$</td>
<td>$n^2 \log n$</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>$n + \log n$</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>$\log n$</td>
<td>$\sqrt{n}$</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>$\ln n$</td>
<td>$\log n$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>$\log(n + 1)$</td>
<td>$\log n$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>$\log \log n$</td>
<td>$\log n$</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$10^n$</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>$n^m$</td>
<td>$m^n$</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>$\cos(n\pi/2)$</td>
<td>$\sin(n\pi/2)$</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$(n \cos n)^2$</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Example

\[ f(n) = \sqrt{n} \quad g(n) = \log n. \quad f(n) + g(n) \]

\[
\lim_{n \to \infty} \frac{g(n)}{f(n)} = \lim_{n \to \infty} \frac{\log n}{\sqrt{n}} \\
= \lim_{n \to \infty} \frac{1/n}{1/(2\sqrt{n})}, \quad \text{(L’Hôpital’s rule)} \\
= \lim_{n \to \infty} 2/\sqrt{n} \\
= 0.
\]

\[ f(n) + g(n) = O(f(n)) \\
= O(\sqrt{n}). \]

\[ n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n. \quad \text{Stirling's approximation} \]
Big-Oh Rules

Theorem

Consider polynomial \( f(n) = \sum_{i=0}^{m} a_i n^i \) where \( a_m > 0 \).

Then \( f(n) = O(n^m) \). i.e.,

1. Drop lower-order terms
2. Drop constant factors

- Use the smallest possible class of functions
  - Say “2n is \( O(n) \)” instead of “2n is \( O(n^2) \)”

- Use the simplest expression of the class
  - Say “3n + 5 is \( O(n) \)” instead of “3n + 5 is \( O(3n) \)”
Big-Oh Rules-2

- $\log^k n = \mathcal{O}(n)$ for any constant $k \in \mathbb{Z}^+$
- $f(n) = \mathcal{O}(f(n))$
- $c \cdot \mathcal{O}(f(n)) = \mathcal{O}(f(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(f(n)) = \mathcal{O}(f(n))$
- $\mathcal{O}(\mathcal{O}(f(n))) = \mathcal{O}(f(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
Relatives of Big-Oh

- **big-Omega**
  - $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

- **big-Theta**
  - $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$
Intuition for Asymptotic Notation

**Big-Oh**
- \( f(n) \) is \( O(g(n)) \) if \( f(n) \) is asymptotically less than or equal to \( g(n) \)

**big-Omega**
- \( f(n) \) is \( \Omega(g(n)) \) if \( f(n) \) is asymptotically greater than or equal to \( g(n) \)

**big-Theta**
- \( f(n) \) is \( \Theta(g(n)) \) if \( f(n) \) is asymptotically equal to \( g(n) \)
Example Uses of the Relatives of Big-Oh

- **$5n^2$ is $\Omega(n^2)$**
  
  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $O(n^2)$**
  
  $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) < c \cdot g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $\Theta(n^2)$**
  
  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$
Comparison of Orders

\[ O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) \]

<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>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>4,096</td>
<td>65,536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1,024</td>
<td>32,768</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>64</td>
<td>384</td>
<td>4,096</td>
<td>262,144</td>
<td>18,446,744,073,709,600,000</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>128</td>
<td>896</td>
<td>16,384</td>
<td>2,097,152</td>
<td>340,282,366,920,936,000,000,000,000,000,000,000,000,000,000</td>
</tr>
</tbody>
</table>

Graph showing the comparison of orders with functions: \( \log n \), \( n \), \( n \log n \), \( n^2 \), \( n^3 \), and \( 2^n \) for various values of \( n \).
Comparison of Orders

\[ O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) \]
O(n) Analysis of Running Time

Example

```c
int findMaximum(int[] a) {
    int result = a[0];
    for (int i = 1; i < a.length; ++i) {
        if (result < a[i]) {
            result = a[i];
        }
    }
    return result;
}
```

Worst-case running time if statement 5 executes all the time...

Best-case running time if statement 5 never executes...

On-the-average running time if statement 5 executes half of the time????
## Algorithm Complexity as a function of 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>Prime factorization</td>
<td>( n )-digit number</td>
<td>division</td>
</tr>
<tr>
<td>Evaluating a polynomial</td>
<td>polynomial of degrees ( n )</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>

- A comparison-based algorithm for searching or sorting a list is based on
  - making comparisons involving list elements
  - then making decisions based on these comparisons.
Quiz-

\[ T(n) = 2T(n - 1) + n, \quad n \geq 0 \]
Quiz- Solution

\[ T(n) = 2T(n-1) + n, \quad n > 0 \]
\[ = 2(2T(n-2) + n-1) + n \]
\[ = 2(2(2T(n-3) + n-2) + n-1) + n \]
\[ \vdots \]
\[ = 2^k T(n-k) + \sum_{i=0}^{k-1} (n-i)2^i \]
\[ \vdots \]
\[ = 2^n T(0) + \sum_{i=0}^{n-1} (n-i)2^i, \quad (n-k) = 0 \]
\[ = 2^n + 2 \cdot 2^n - n - 2 \]
\[ = 3 \cdot 2^n - n - 2. \]