Advanced Algorithmics (6EAP)
MTAT.03.238
courses.cs.ut.ee
Order of growth... maths

Jaak Vilo
2018 fall
Program execution on input of size n

• How many steps/cycles a processor would need to do

• How to relate algorithm execution to nr of steps on input of size n? \( f(n) \)

• e.g. \( f(n) = n + n*(n-1)/2 + 17 n + n*log(n) \)
What happens in infinity?

- Faster computer, larger input?

- **Bad algorithm on fast computer** will be outcompeted by good algorithm on slow...
## Big-Oh notation classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Informal</th>
<th>Intuition</th>
<th>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) \in \mathcal{O}(g(n)) )</td>
<td>f is dominated by g</td>
<td>Strictly below</td>
<td>&lt;</td>
</tr>
<tr>
<td>( f(n) \in \Theta(g(n)) )</td>
<td>Bounded from above and below</td>
<td>“equal to”</td>
<td>=</td>
</tr>
<tr>
<td>( f(n) \in \Omega(g(n)) )</td>
<td>Bounded from below</td>
<td>Lower bound</td>
<td>≥</td>
</tr>
<tr>
<td>( f(n) \in \omega(g(n)) )</td>
<td>f dominates g</td>
<td>Strictly above</td>
<td>&gt;</td>
</tr>
</tbody>
</table>
Mathematical Background
Justification

• As engineers, you will not be paid to say:

    Method A is *better* than Method B

or

    Algorithm A is *faster* than Algorithm B

• Such descriptions are said to be *qualitative* in nature; from the OED:

**qualitative, a. a** Relating to, connected or concerned with, quality or qualities. Now usually in implied or expressed opposition to quantitative.
Mathematical Background

Justification

• Business decisions cannot be based on qualitative statements:
  – Algorithm A could be *better* than Algorithm B, but Algorithm A would require three person weeks to implement, test, and integrate while Algorithm B has already been implemented and has been used for the past year
  – there are circumstances where it may beneficial to use Algorithm A, but not based on the word *better*
Mathematical Background

Justification

• Thus, we will look at a *quantitative* means of describing data structures and algorithms

• From the OED:

  **quantitative, a.** Relating to or concerned with quantity or its measurement; that assesses or expresses quantity. In later use freq. contrasted with *qualitative*.

We need to quantify program behavior ... theoretically and in practice
Program time and space complexity

• Time: count nr of elementary calculations/operations during program execution

• Space: count amount of memory (RAM, disk, tape, flash memory, …)
  – usually we do not differentiate between cache, RAM, …
  – In practice, for example, random access on tape impossible
• Program 1.17

```c
float Sum(float *a, const int n)
{
    float s = 0;
    for (int i=0; i<n; i++)
        s += a[i];
    return s;
}
```

– The instance characteristic is $n$.
– Since $a$ is actually the address of the first element of $a[]$, and $n$ is passed by value, the space needed by `Sum()` is constant ($S_{\text{sum}}(n)=1$).
Program 1.18

```c
float RSum(float *a, const int n)
{
    if (n <= 0) return 0;
    else return (RSum(a, n-1) + a[n-1]);
}
```

– Each call requires at least 4 words
  • The values of n, a, return value and return address.
– The depth of the recursion is n+1.
  • The stack space needed is 4(n+1).
Input size = n

• Input size usually denoted by \( n \)
• Time complexity function = \( f(n) \)
  – array[1..n]
  – e.g. \( 4*n + 3 \)

• Graph: \( n \) vertices, \( m \) edges \( f(n,m) \)
1.4 Fibonacci numbers

Let us consider another example where the choice of a proper algorithm leads to an even more dramatic improvement in efficiency.

The sequence of Fibonacci numbers $F_0, F_1, \ldots$ is defined by the well-known recursion formula:

$$F_n = \begin{cases} 
0, & \text{if } n = 0, \\
1, & \text{if } n = 1, \\
F_{n-1} + F_{n-2}, & \text{if } n \geq 2.
\end{cases}$$

This sequence arises in a natural way in countless applications. The numbers grow at an exponential rate: $F_n \approx 2^{0.694n}$. (Exercise.)
It is tempting to use the definition of Fibonacci numbers directly as the basis of a simple recursive method for computing them:

Algorithm 1: First algorithm for Fibonacci numbers

1. function FIB1(n)
2. if $n = 0$ then return 0
3. if $n = 1$ then return 1
4. else return $FIB1(n-1) + FIB1(n-2)$

This is, however, not a good idea, because the computation of $FIB1(n)$ requires time proportional to the value of the $F_n$ itself. (Verify this!)
The reason for the inefficiency is that the algorithm recomputes most of the values several (in fact, exponentially many) times:

Figure: Computation graph for Fibonacci numbers.
The recomputations can, however, be easily avoided by computing the values iteratively “bottom-up” and tabulating them:

**Algorithm 2:** Improved algorithm for Fibonacci numbers

1. function \texttt{Fib2}(n)
2. if \( n = 0 \) then return 0
3. else
4. Introduce auxiliary array \( F[0\ldots n] \)
5. \( F[0] \leftarrow 0; F[1] \leftarrow 1 \)
6. for \( i \leftarrow 2 \) to \( n \) do
7. \hspace{1em} \( F[i] \leftarrow F[i - 1] + F[i - 2] \)
8. end
9. return \( F[n] \)
10. end

The computation time is now just \( O(n) \) — a huge improvement!
2.1 Analysis of algorithms: basic notions

- Generally speaking, an algorithm $A$ computes a mapping from a potentially infinite set of inputs (or instances) to a corresponding set of outputs (or solutions).
- Denote by $T(x)$ the number of elementary operations that $A$ performs on input $x$ and by $|x|$ the size of an input instance $x$.
- Denote by $T(n)$ also the worst-case time that algorithm $A$ requires on inputs of size $n$, i.e.,

$$T(n) = \max \{ T(x) : |x| = n \}. $$
2.2 Analysis of iterative algorithms

To illustrate the basic worst-case analysis of iterative algorithms, let us consider yet another well-known (but bad) sorting method.

**Algorithm 3: The insertion sort algorithm**

```
1 function INSERTSORT (A[1 ... n])
2 for i ← 2 to n do
3    a ← A[i]; j ← i - 1
4    while j > 0 and a < A[j] do
5       A[j + 1] ← A[j]; j ← j - 1
6     end
7     A[j + 1] ← a
8 end
```
Analysis of insertion sort

Denote: $T_{k-l} = \text{the complexity of a single execution of lines } k \text{ thru } l$. Then:

\[
\begin{align*}
T_5(n, i, j) & \leq c_1 \\
T_{4-6}(n, i) & \leq c_2 + (i - 1)c_1 \\
T_{3-7}(n, i) & \leq c_3 + c_2 + (i - 1)c_1 \\
T_{2-8}(n) & \leq c_4 + \sum_{i=2}^{n}(c_3 + c_2 + (i - 1)c_1) \\
& = c_4 + (n - 1)(c_3 + c_2) + c_1 \sum_{i=2}^{n}(i - 1) \\
& \leq \text{const} \cdot n + c_1 \cdot \frac{1}{2}n(n - 1)
\end{align*}
\]

Thus $T(n) = T_{2-8}(n) = O(n^2)$. 
Basic analysis rules

Denote $T[P]$ = the time complexity of an algorithm segment $P$.

- $T[x \leftarrow e] = \text{constant}$, $T[\text{read } x] = \text{constant}$, $T[\text{write } x] = \text{constant}$.
- $T[S_1; S_2; \ldots; S_k] = T[S_1] + \cdots + T[S_k] = O(\max\{T[S_1], \ldots, T[S_k]\})$
- $T[\text{if } P \text{ then } S_1 \text{ else } S_2] =$
  \[
  \begin{cases}
  T[P] + T[S_1] & \text{if } P = \text{true} \\
  T[P] + T[S_2] & \text{if } P = \text{false}
  \end{cases}
  \]
- $T[\text{while } P \text{ do } S] =$
  
  $T[P] + (\text{number of times } P = \text{true}) \cdot (T[S] + T[P])$

In analysing nested loops, proceed from innermost out. Control variables of outer loops enter as parameters in the analysis of inner loops.
• So, we may be able to count a nr of operations needed

• What do we do with this knowledge?
The image shows a graphical representation of two functions, $2x^2$ and $100x$, plotted on a graph. The x-axis ranges from 0 to 100, and the y-axis ranges from 0 to 20000. The graph demonstrates the exponential growth of $2x^2$ compared to the linear growth of $100x$. Additionally, there is a terminal window showing the gnuplot command used to generate the graph: `plot [1:100] 2*x**2, 100*x`.
GNUPLOT
Version 4.4 patchlevel 0-rc1
last modified November 2009
System: MS-Windows 32 bit

Thomas Williams, Colin Kelley and many others

gnuplot home: http://www.gnuplot.info
faq, bugs, etc: type "help seeking-assistance"
immediate help: type "help"
plot window: hit 'h'

Terminal type set to 'windows'
gnuplot> plot x*x
gnuplot> plot [0:1000] x*x
gnuplot> plot [0:1000] x*x, 10*x
gnuplot> plot [0:100] x*x, 10*x
gnuplot> plot [0:100] x*x, 100*x
gnuplot> plot [0:100] x*x, 200*x
gnuplot> plot [0:100] x, x*x, x*x*x
gnuplot> plot [0:100] x, x*x
gnuplot> plot [0:100] 1000*x, x*x
gnuplot> plot [0:100]
$n^2$

$100 \cdot n \log n$
$n^2$

$100 \, n \, \log(n)$

$n = 1000$
0.00001\*n^2
100 n \log n

n = 10,000,000
\[ 0.00001 * n^2 \]
\[ 100 \cdot n \cdot \log n \]

\( n = 2,000,000,000 \)
Logscale y
logscale x
logscale y
• plot [1:10] 0.01*x*x, 5*log(x), x*log(x)/3
Algorithm analysis goal

- What happens in the “long run”, increasing $n$

- Compare $f(n)$ to reference $g(n)$ (or $f$ and $g$)

- At some $n_0 > 0$, if $n > n_0$, always $f(n) < g(n)$
Asymptotic notation

\textit{O-notation (upper bounds)}:

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

\textbf{Example}: $2n^2 = O(n^3)$ ($c = 1$, $n_0 = 2$)

\textit{funny, “one-way” equality}

\textit{functions, not values}
Set definition of O-notation

\[ O(g(n)) = \{ f(n) : \text{there exist constants } c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \} \]

**Example:** \( 2n^2 \in O(n^3) \)
\( \Omega \)-notation (lower bounds)

\( O \)-notation is an upper-bound notation. It makes no sense to say \( f(n) \) is at least \( O(n^2) \).

\[
\Omega(g(n)) = \{ f(n) : \text{there exist constants } c > 0, n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}
\]

Example: \( \sqrt{n} = \Omega(\log n) \) \( (c = 1, n_0 = 16) \)
\( \Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) \)

**Example:** \( \frac{1}{2} n^2 - 2n = \Theta(n^2) \)
Figure 2.1 Graphic examples of the $\Theta$, $O$, and $\Omega$ notations. In each part, the value of $n_0$ shown is the minimum possible value; any greater value would also work.
-notation and $\omega$-notation

$O$-notation and $\Omega$-notation are like $\leq$ and $\geq$.

$o$-notation and $\omega$-notation are like $<$ and $>$. 

$$o(g(n)) = \{ f(n) : \text{for any constant } c > 0, \text{there is a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0 \}$$

**Example:** $2n^2 = o(n^3) \quad (n_0 = 2/c)$
**o-notation and ω-notation**

*O*-notation and *Ω*-notation are like ≤ and ≥. *o*-notation and *ω*-notation are like < and >.

\[ \omega(g(n)) = \{ f(n) : \text{for any constant } c > 0, \text{ there is a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0 \} \]

**Example:** \[ \sqrt{n} = \omega(\log n) \quad (n_0 = 1 + 1/c) \]
Macro substitution

Convention: A set in a formula represents an anonymous function in the set.

Example: \( f(n) = n^3 + O(n^2) \)
means
\( f(n) = n^3 + h(n) \)
for some \( h(n) \in O(n^2) \).
Dominant terms only...

• Essentially, we are interested in the largest (dominant) term only...

• When this grows large enough, it will “overshadow” all smaller terms
Theorem 1.2

If \( f(n) = a_m n^m + \ldots + a_1 n + a_0 \), then \( f(n) = O(n^m) \).

Proof:

\[
f(n) = \sum_{i=0}^{m} a_i n^i \leq \sum_{i=0}^{m} |a_i| n^i \]

\[
\leq n^m \sum_{i=0}^{m} |a_i| n^{i-m} \]

\[
\leq n^m \sum_{i=0}^{m} |a_i|, \text{ for } n \geq 1. \]

Therefore, let \( c = \sum_{i=0}^{m} |a_i| \), \( n_0 = 1 \), we have

\[
f(n) \leq cn^m, \text{ for } n \geq n_0. \text{ Thes, } f(n) = O(n^m). \]
Asymptotic Analysis

• Given any two functions $f(n)$ and $g(n)$, we will restrict ourselves to:
  – polynomials with positive leading coefficient
  – exponential and logarithmic functions
• These functions $\rightarrow \infty$ as $n \rightarrow \infty$
• We will consider the limit of the ratio:

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

• If the two functions $f(n)$ and $g(n)$ describe the run times of two algorithms, and

$$0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

that is, the limit is a constant, then we can always run the slower algorithm on a faster computer to get similar results.
Asymptotic Analysis

• To formally describe equivalent run times, we will say that 
  \( f(n) = \Theta(g(n)) \) if

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

• Note: this is not equality – it would have been better if it said 
  \( f(n) \in \Theta(g(n)) \) however, someone picked =
Asymptotic Analysis

• We are also interested if one algorithm runs either asymptotically slower or faster than another

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

• If this is true, we will say that \( f(n) = \Theta(g(n)) \)
Asymptotic Analysis

• If the limit is zero, \(i.e.,\)

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

then we will say that \(f(n) = o(g(n))\)

• This is the case if \(f(n)\) and \(g(n)\) are polynomials where \(f\) has a lower degree
Asymptotic Analysis

• To summarize:

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

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

\[ f(n) = O(g(n)) \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty \]
Asymptotic Analysis

- We have one final case:

\[
\begin{align*}
    f(n) &= \omega(g(n)) & \lim_{n \to \infty} \frac{f(n)}{g(n)} &= \infty \\
    f(n) &= \Theta(g(n)) & 0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty \\
    f(n) &= o(g(n)) & \lim_{n \to \infty} \frac{f(n)}{g(n)} &= 0
\end{align*}
\]
Asymptotic Analysis

• Graphically, we can summarize these as follows:

We say
\[ f(n) = \begin{cases} 
O(g(n)) & \text{if } \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \\
\Omega(g(n)) & \text{if } \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \\
o(g(n)) & \text{if } \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 < c < \infty \\
\Theta(g(n)) & \text{if } \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \\
\omega(g(n)) & \text{if } \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 
\end{cases} \]
Asymptotic Analysis

• All of
  \[ n^2, 100000 n^2 - 4 n + 19, n^2 + 1000000, \]
  \[ 323 n^2 - 4 n \ln(n) + 43 n + 10, 42n^2 + 32, \]
  \[ n^2 + 61 n \ln^2(n) + 7n + 14 \ln^3(n) + \ln(n) \]
  are big-\(\Theta\) of each other

• E.g., \(42n^2 + 32 = \Theta( 323 n^2 - 4 n \ln(n) + 43 n + 10 )\)
Asymptotic Analysis

• We will focus on these

\( \Theta(1) \) constant
\( \Theta(\ln(n)) \) logarithmic
\( \Theta(n) \) linear
\( \Theta(n \ln(n)) \) “n–log–n”
\( \Theta(n^2) \) quadratic
\( \Theta(n^3) \) cubic

\( 2^n, e^n, 4^n, ... \) exponential

\( O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) \)
Growth of functions

• See Chapter “Growth of Functions” (CLRS)


 Gnuplot (window id : 0) 

\[ \text{log}(\text{int}(x)) \]  
\[ x^{\text{log}(x)} \]
Figure 0

log( int(x)! )

x*log(x)
\[ \lg n \quad = \quad \log_2 n \quad \text{(binary logarithm)}, \]
\[ \ln n \quad = \quad \log_e n \quad \text{(natural logarithm)}, \]
\[ \lg^k n \quad = \quad (\lg n)^k \quad \text{(exponentiation)}, \]
\[ \lg \lg n \quad = \quad \lg (\lg n) \quad \text{(composition)}. \]
Logarithms


Logarithms and Log Properties

Definition

\[ y = \log_b x \text{ is equivalent to } x = b^y \]

Example

\[ \log_5 125 = 3 \text{ because } 5^3 = 125 \]

Special Logarithms

\[ \ln x = \log_e x \text{ natural log} \]

\[ \log x = \log_{10} x \text{ common log} \]

where \( e = 2.718281828 \ldots \)

Logarithm Properties

\[ \log_b b = 1 \quad \log_b 1 = 0 \]

\[ \log_b b^x = x \quad b^{\log_b x} = x \]

\[ \log_b (x^r) = r \log_b x \]

\[ \log_b (xy) = \log_b x + \log_b y \]

\[ \log_b \left(\frac{x}{y}\right) = \log_b x - \log_b y \]

The domain of \( \log_b x \) is \( x > 0 \)
Change of base $a \rightarrow b$

$$\log_b x = \frac{1}{\log_a b} \log_a x$$

$$\log_b x = \Theta(\log_a x)$$
## Big-Oh notation classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Informal</th>
<th>Intuition</th>
<th>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(n) \in O(\ g(n) \ )$</td>
<td>$f$ is dominated by $g$</td>
<td>Strictly below</td>
<td>$&lt;$</td>
</tr>
<tr>
<td>$f(n) \in \Theta(\ g(n) \ )$</td>
<td>Bounded from above and below</td>
<td>“equal to”</td>
<td>$=$</td>
</tr>
<tr>
<td>$f(n) \in \Omega(\ g(n) \ )$</td>
<td>Bounded from below</td>
<td>Lower bound</td>
<td>$\geq$</td>
</tr>
<tr>
<td>$f(n) \in \omega(\ g(n) \ )$</td>
<td>$f$ dominates $g$</td>
<td>Strictly above</td>
<td>$&gt;$</td>
</tr>
</tbody>
</table>
## Family of Bachmann–Landau notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Name</th>
<th>Intuition</th>
<th>As ( n \to \infty ), eventually...</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) \in \mathcal{O}(g(n)) )</td>
<td>Big Omicron; Big O; Big Oh</td>
<td>( f ) is bounded above by ( g ) (up to constant factor) asymptotically</td>
<td>(</td>
<td>f(n)</td>
</tr>
<tr>
<td>( f(n) \in \Omega(g(n)) )</td>
<td>Big Omega</td>
<td>( f ) is bounded below by ( g ) (up to constant factor) asymptotically</td>
<td>( f(n) \geq g(n) \cdot k ) for some positive ( k )</td>
<td>( \exists k &gt; 0, \exists n_0, \forall n &gt; n_0, g(n) \cdot k \leq f(n) )</td>
</tr>
<tr>
<td>( f(n) \in \Theta(g(n)) )</td>
<td>Big Theta</td>
<td>( f ) is bounded both above and below by ( g ) asymptotically</td>
<td>( g(n) \cdot k_1 \leq f(n) \leq g(n) \cdot k_2 ) for some positive ( k_1, k_2 )</td>
<td>( \exists k_1 &gt; 0, \exists k_2 &gt; 0, \exists n_0, \forall n &gt; n_0, g(n) \cdot k_1 \leq f(n) \leq g(n) \cdot k_2 )</td>
</tr>
<tr>
<td>( f(n) \in o(g(n)) )</td>
<td>Small Omicron; Small O; Small Oh</td>
<td>( f ) is dominated by ( g ) asymptotically</td>
<td>(</td>
<td>f(n)</td>
</tr>
<tr>
<td>( f(n) \in \omega(g(n)) )</td>
<td>Small Omega</td>
<td>( f ) dominates ( g ) asymptotically</td>
<td>( f(n) \geq g(n) \cdot k ) for every ( k )</td>
<td>( \forall k &gt; 0, \exists n_0, \forall n &gt; n_0, g(n) \cdot k \leq f(n) )</td>
</tr>
<tr>
<td>( f(n) \sim g(n) )</td>
<td>on the order of; &quot;twiddles&quot;</td>
<td>( f ) is equal to ( g ) asymptotically</td>
<td>( f(n)/g(n) \to 1 )</td>
<td>( \forall \varepsilon &gt; 0, \exists n_0, \forall n &gt; n_0, \left</td>
</tr>
</tbody>
</table>
Problems 3-3: Ordering by asymptotic growth rates

a. Rank the following functions by order of growth; that is, find an arrangement \( g_1, g_2, \ldots, g_{30} \) of the functions satisfying \( g_1 = (g_{\ 2}), g_2 = (g_{\ 3}), \ldots, g_{29} = (g_{\ 30}) \). Partition your list into equivalence classes such that \( f(n) \) and \( g(n) \) are in the same class if and only if \( f(n) = \Theta(g(n)) \).

\[
\begin{aligned}
\lg(\lg^* n) & \quad 2^{\lg^* n} & \quad (\sqrt{2})^{\lg n} & \quad n^2 & \quad n! & \quad (\lg n)! \\
\left(\frac{3}{2}\right)^n & \quad n^3 & \quad \lg^2 n & \quad \lg(n!) & \quad 2^{2^n} & \quad n^{1/\lg n} \\
\ln \ln n & \quad \lg^* n & \quad n \cdot 2^n & \quad n^\lg \lg n & \quad \ln n & \quad 1 \\
2^{\lg n} & \quad (\lg n)^{\lg n} & \quad e^n & \quad 4^{\lg n} & \quad (n + 1)! & \quad \sqrt{\lg n} \\
\lg^* (\lg n) & \quad 2^{\sqrt{2\lg n}} & \quad n & \quad 2^n & \quad n \lg n & \quad 2^{2^n + 1}
\end{aligned}
\]
Functional iteration

We use the notation $f^{(i)}(n)$ to denote the function $f(n)$ iteratively applied $i$ times to an initial value of $n$. Formally, let $f(n)$ be a function over the reals. For nonnegative integers $i$, we recursively define

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0, \\ f(f^{(i-1)}(n)) & \text{if } i > 0. \end{cases}$$

For example, if $f(n) = 2n$, then $f^{(0)}(n) = 2^n$.

The iterated logarithm function

We use the notation $\lg^* n$ (read "log star of $n$") to denote the iterated logarithm, which is defined as follows. Let $\lg^{(i)}n$ be as defined above, with $f(n) = \lg n$. Because the logarithm of a nonpositive number is undefined, $\lg^{(i)}n$ is defined only if $\lg^{(i-1)}n > 0$. Be sure to distinguish $\lg^{(i)}n$ (the logarithm function applied $i$ times in succession, starting with argument $n$) from $\lg^n n$ (the logarithm of $n$ raised to the $i$th power). The iterated logarithm function is defined as

$$\lg^* n = \min \{i = 0 : \lg^{(i)}n = 1\}.$$
The iterated logarithm is a very slowly growing function:

\[
\begin{align*}
\lg^* 2 &= 1, \\
\lg^* 4 &= 2, \\
\lg^* 16 &= 3, \\
\lg^* 65536 &= 4, \\
\lg^*(2^{65536}) &= 5.
\end{align*}
\]

Since the number of atoms in the observable universe is estimated to be about $10^{80}$, which is much less than $2^{65536}$, we rarely encounter an input size $n$ such that $\lg^* n > 5$. 
googol
How much time does sorting take?

- Comparison-based sort: $A[i] \leq A[j]$
  - **Upper bound** – current best-known algorithm
  - **Lower bound** – theoretical “at least” estimate
  - If they are equal, we have theoretically optimal solution
Simple sort

for i=2..n
  for j=i ; j>1 ; j--
      then swap( A[j], A[j-1] )
    else next i
The divide-and-conquer design paradigm

1. *Divide* the problem (instance) into subproblems.
2. *Conquer* the subproblems by solving them recursively.
3. *Combine* subproblem solutions.
Merge sort

**Merge-Sort**(*A*,*p*,*r*)

**if** *p*<*r* **then** *q* = (*p*+*r*)/2

**Merge-Sort**(*A*, *p*, *q*)

**Merge-Sort**(*A*, *q*+1,*r*)

**Merge**(*A*, *p*, *q*, *r*)

It was invented by **John von Neumann** in 1945.
Example

• Applying the merge sort algorithm:
Wikipedia / viz.
Divide and conquer

Quicksort an $n$-element array:

1. **Divide**: Partition the array into two subarrays around a **pivot** $x$ such that elements in lower subarray $\leq x \leq$ elements in upper subarray.

   \[
   \begin{array}{ccc}
   \leq x & x & \geq x \\
   \end{array}
   \]

2. **Conquer**: Recursively sort the two subarrays.

3. **Combine**: Trivial.

   **Key**: *Linear-time partitioning subroutine.*
Pseudocode for quicksort

**QUICKSORT**(*A, p, r*)

if \( p < r \)

then \( q \leftarrow \text{PARTITION}(A, p, r) \)

**QUICKSORT**(*A, p, q-1*)

**QUICKSORT**(*A, q+1, r*)

Initial call: **QUICKSORT**(*A, 1, n*)
Partitioning subroutine

\[ \text{Partition}(A, p, q) \rightarrow A[p..q] \]

\[ x \leftarrow A[p] \quad \text{pivot} = A[p] \]
\[ i \leftarrow p \]
\[ \text{for} \ j \leftarrow p + 1 \text{ to } q \]
\[ \quad \text{do if } A[j] \leq x \]
\[ \quad \text{then } i \leftarrow i + 1 \]
\[ \quad \text{exchange } A[i] \leftrightarrow A[j] \]
\[ \text{exchange } A[p] \leftrightarrow A[i] \]
\[ \text{return } i \]

**Invariant:**

\[ \begin{array}{c|c|c|c|}
  x & \leq x & \geq x & ? \\
  p & i & j & q 
\end{array} \]

Running time

\[ = \mathcal{O}(n) \text{ for } n \text{ elements.} \]
Wikipedia / “video”
Conclusions

• Algorithm complexity deals with the behavior in the long-term
  – worst case -- typical
  – average case -- quite hard
  – best case -- bogus, “cheating”

• In practice, long-term sometimes not necessary
  – E.g. for sorting 20 elements, you don’t need fancy algorithms...
Breaking the Coppersmith-Winograd barrier

Virginia Vassilevska Williams
UC Berkeley and Stanford University

Abstract

We develop new tools for analyzing matrix multiplication constructions similar to the Coppersmith-Winograd construction, and obtain a new improved bound on $\omega < 2.3727$.

1 Introduction

The product of two matrices is one of the most basic operations in mathematics and computer science. Many other essential matrix operations can be efficiently reduced to it, such as Gaussian elimination, LUP decomposition, the determinant or the inverse of a matrix [1]. Matrix multiplication is also used as a subroutine in many computational problems that, on the face of it, have nothing to do with matrices. As a small sample illustrating the variety of applications, there are faster algorithms relying on matrix multiplication for graph transitive closure (see e.g. [1]), context free grammar parsing [20], and even learning juntas [13].

Until the late 1960s it was believed that computing the product $C$ of two $n \times n$ matrices requires essentially a cubic number of operations, as the fastest algorithm known was the naive algorithm which indeed runs in $O(n^3)$ time. In 1969, Strassen [19] excited the research community by giving the first subcubic time algorithm for matrix multiplication, running in $O(n^{2.805})$ time. This amazing discovery spawned a long line of research which gradually reduced the matrix multiplication exponent $\omega$ over time. In 1978, Pan [14] showed $\omega < 2.796$. The following year, Bini et al. [4] introduced the notion of border rank and obtained $\omega < 2.78$. Schönhage [17] generalized this notion in 1981, proved his $\tau$-theorem (also called the asymptotic sum inequality), and showed that $\omega < 2.548$. In the same paper, combining his work with ideas by Pan, he also showed $\omega < 2.522$. The following year, Romani [15] found that $\omega < 2.517$. The first result to break 2.5 was by Coppersmith and Winograd [9] who obtained $\omega < 2.496$. In 1985, Strassen introduced his laser method which allowed for an entirely new attack on the matrix multiplication problem. He also decreased the bound to $\omega < 2.479$. Three years later, Coppersmith and Winograd [10] combined Strassen’s technique with a novel form of analysis based on large sets avoiding arithmetic progressions and obtained the famous bound of $\omega < 2.376$ which has remained unchanged for more than twenty years.

In 2003, Cohn and Umans [8] introduced a new, group-theoretic framework for designing and analyzing matrix multiplication algorithms. In 2005, together with Kleinberg and Szegedy [7], they obtained several