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) ∈ o(g(n))</td>
<td>f is dominated by g</td>
<td>Strictly below</td>
<td>&lt;</td>
</tr>
<tr>
<td>f(n) ∈ O(g(n))</td>
<td>Bounded from above</td>
<td>Upper bound</td>
<td>≤</td>
</tr>
<tr>
<td>f(n) ∈ Θ(g(n))</td>
<td>Bounded from above and below</td>
<td>“equal to”</td>
<td>=</td>
</tr>
<tr>
<td>f(n) ∈ Ω(g(n))</td>
<td>Bounded from below</td>
<td>Lower bound</td>
<td>≥</td>
</tr>
<tr>
<td>f(n) ∈ ω(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_{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: \( F_n \approx 2^n \text{ for } n \to \infty \).

(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 \( \text{Fib}(n) \)
2. if \( n = 0 \) then return 0
3. if \( n = 1 \) then return 1
4. else return \( \text{Fib}(n-1) + \text{Fib}(n-2) \)

This is, however, not a good idea, because the computation of \( \text{Fib}(n) \) requires time proportional to the value of the \( F_n \) itself. (Verify this!)

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 \( \text{Fib}(n) \)
2. if \( n = 0 \) then return 0
3. else
4. introduce auxiliary array \( F[0 \ldots n] \)
5. \( F[0] = 0; F[1] = 1 \)
6. for \( i = 2 \) to \( n \) do
7. \( F[i] = F[i-1] + F[i-2] \)
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 \( \text{INSERTSORT}(A[1 \ldots n]) \)
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: \( T_{k,l} \) = the complexity of a single execution of lines \( k \) thru \( l \). Then:
\[
\begin{align*}
T_{2}(n,1) &\leq c_1 \\
T_{4}(n,1) &\leq c_2 + (i - 1)c_0 \\
T_{5}(n,1) &\leq c_3 + c_0 + (i - 1)c_1 \\
T_{2,n}(0) &\leq c_4 + \sum_{i=2}^{n}(c_0 + c_2 + (i - 1)c_1) \\
&= c_4 + (n - 1)(c_0 + c_2) + c_1 \sum_{i=2}^{n}(i - 1) \\
&\leq \text{const} \cdot n + c_1 \cdot \frac{3}{2}(n - 1) \\
\end{align*}
\]
Thus \( T(n) = \Theta(n^2) \).
• So, we may be able to count a nr of operations needed

• What do we do with this knowledge?

Basic analysis rules
Denote $T[P]$ the time complexity of an algorithm segment $P$.

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

In analysing nested loops, proceed from innermost out. Control variables of outer loops enter as parameters in the analysis of inner loops.
\[
0.000001 n^2 \\
100 n \log n
\]

\[
n = 10,000,000
\]

\[
0.000001 n^2 \\
100 n \log n
\]

\[
n = 2,000,000,000
\]

- 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

$O$-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 $n \geq n_0$.

**Example:** $2n^2 = O(n^3)$ ($c = 1$, $n_0 = 2$)

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:** $\sqrt{n} = \Omega(\log n)$ ($c = 1$, $n_0 = 16$)
**Θ-notations (tight bounds)**

\[ \Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) \]

**Example:** \[ \frac{1}{2} n^2 - 2n = \Theta(n^2) \]

---

**Θ, O, and Ω**

Figure 2.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of \( n_0 \) shown is the minimum possible value; any greater value would also work.

---

**ο-notation and ω-notation**

O-notation and Ω-notation are like \( \leq \) and \( \geq \). ο-notation and ω-notation are like \( < \) and \( > \).

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

**Example:** \[ 2n^2 = o(n^3) \quad (n_0 = 2/c) \]

---

**ο-notation and ω-notation**

O-notation and Ω-notation are like \( \leq \) and \( \geq \). ο-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 \( f(n) = a_n n^n + \ldots + a_1 n + a_0 \), then \( f(n) = O(n^n) \).

Proof:

\[
  f(n) = \sum_{r=0}^\infty a_r n^r \leq \sum_{r=0}^\infty |a_r| n^r \\
  \leq n^m \sum_{r=0}^\infty |a_r| n^{r-m} \\
  \leq n^m \sum_{r=0}^\infty |a_r|, \text{ for } n \geq 1.
\]

Therefore, let \( c = \sum_{r=0}^\infty |a_r| n_0 = 1 \), we have

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

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 function \( f(n) \) 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 \( f(n) = O(g(n)) \)

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

Asymptotic Analysis
• Graphically, we can summarize these as follows:
  We say \[ f(n) = \frac{O(g(n))}{o(g(n))} = \frac{\Omega(g(n))}{\omega(g(n))} = \Theta(g(n)) \]
  if \[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} 
0 & 0 < \varepsilon < \infty \\
0 & \infty 
\end{cases} \]

Asymptotic Analysis
• All of
  \[ n^2 \quad 100000 \quad n^2 - 4n + 19 \quad n^2 + 1000000 \]
  \[ 323n^2 - 4n \ln(n) + 43n + 10 \quad 42n^2 + 32 \]
  \[ n^2 + 61n \ln^2(n) + 7n + 14 \ln^3(n) + \ln(n) \]
  are big-\(\Theta\) of each other
  \[ \text{E.g.}, \quad 42n^2 + 32 = \Theta(323n^2 - 4n \ln(n) + 43n + 10) \]

Asymptotic Analysis
• We will focus on these
  \[ \Theta(1) \quad \text{constant} \]
  \[ \Theta(n) \quad \text{linear} \]
  \[ \Theta(n \ln(n)) \quad \text{“n–log–n”} \]
  \[ \Theta(n^2) \quad \text{quadratic} \]
  \[ \Theta(n^3) \quad \text{cubic} \]
  \[ 2^n, \ e^n, \ 4^n, \ldots \quad \text{exponential} \]

Growth of functions
• See Chapter “Growth of Functions” (CLRS) .
Logarithms


\[
\begin{align*}
\lg n &= \log_2 n \quad \text{(binary logarithm)}, \\
\ln n &= \log_e n \quad \text{(natural logarithm)}, \\
\lg^k n &= (\lg n)^k \quad \text{(exponentiation)}, \\
\lg \lg n &= \lg(\lg n) \quad \text{(composition)}.
\end{align*}
\]

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**
\[ \ln x = \log_e x \quad \text{natural log} \]
\[ \log x = \log_{10} x \quad \text{common log} \]
where \( e = 2.718281828 \ldots \)

Logarithm Properties

\[
\begin{align*}
\log_b b &= 1 \\
\log_1 b &= 0 \\
\log_b b^x &= x \\
\log_b a^x &= x \log_b a \\
\log_b (xy) &= \log_b x + \log_b y \\
\log_b \left( \frac{x}{y} \right) &= \log_b x - \log_b y
\end{align*}
\]

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>$f(n) \in O(g(n))$</td>
<td>Bounded from above</td>
<td>Upper bound</td>
<td>$\leq$</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>
</tr>
</thead>
<tbody>
<tr>
<td>$f(n) \in o(g(n))$</td>
<td>Big Omicron; Big O; Big Oh</td>
<td>$f$ is dominated by $g$ asymptotically</td>
</tr>
<tr>
<td>$f(n) \in O(g(n))$</td>
<td>Big Omega</td>
<td>$f$ is bounded above by $g$ asymptotically</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>
</tr>
<tr>
<td>$f(n) \in \Omega(g(n))$</td>
<td>Small Omega</td>
<td>$f$ dominates $g$ asymptotically</td>
</tr>
</tbody>
</table>

### Functional iteration

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

$f^{(0)}(n) = n$  
$f^{(k+1)}(n) = f^{(k)}(f(n))$  

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

### The iterated logarithm function

We use the notation $\log^n$ as a sort of “log-log-log,” to denote the iterated logarithm, which is defined as follows. Let $\log(n) = \log\log(n)$, where $\log$ is the logarithm to base 2. Define $\log_1(n)$ as the integer $k$ such that $2^k \leq n < 2^{k+1}$. Define $\log_{-1}(n) = \log\log(n)$.

The iterated logarithm is defined as:

$\log^0(n) = n$  
$\log^{k+1}(n) = \log(\log^k(n))$  

Thus, $\log^{16}(n)$ is the number of times the logarithm function must be applied to $n$ in order to get a value less than or equal to 1. For example, the maximum number of times the iterated logarithm function must be applied to $2^{100}$ to get a value less than or equal to 1 is $\log^{16}(2^{100}) = 100 \log \log 100 \approx 16$. This is a very slow-growing function.
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

```plaintext
for i=2..n
    for j=i; j>1; j--;
            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.

Example

- Applying the merge sort algorithm:

```plaintext
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.
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 \( \leq x \) ≤ elements in upper subarray \( \geq x \).
2. Conquer: Recursively sort the two subarrays.

Key: Linear-time partitioning subroutine.

Pseudocode for quicksort

\[
\text{QUICKSORT}(A, p, r) \\
\begin{align*}
\text{if } p &< r \\
\quad &\text{then } q \leftarrow \text{PARTITION}(A, p, r) \\
\text{QUICKSORT}(A, p, q-1) &\\
\text{QUICKSORT}(A, q+1, r) &
\end{align*}
\]

Initial call: \( \text{QUICKSORT}(A, 1, n) \)

Partitioning subroutine

\[
\text{PARTITION}(A, p, q) \rightarrow A[p..q] \\
\begin{align*}
&x \leftarrow A[p] \\
&\text{pivot } \leftarrow A[p] \\
&i \leftarrow p \\
&\text{for } j \leftarrow p+1 \text{ to } q \\
&\quad \text{do if } A[j] \leq x \\
&\quad \quad \text{then } i \leftarrow i+1 \\
&\quad \quad \text{exchange } A[i] \leftrightarrow A[j] \\
&\text{return } i
\end{align*}
\]

Running time = \( O(n) \) for \( n \) elements.

Invariant:

\[
\begin{array}{cccccc}
\leq & x & \geq x & j & A[i] & q \\
p & i & & & & \\
\end{array}
\]

Conclusions

- Algorithm complexity deals with the behavior in the long-term
  - worst case
  - average case
  - best case
- In practice, long-term sometimes not necessary
  - E.g. for sorting 20 elements, you don’t need fancy algorithms...
Abstract

We present a tool for analyzing matrix multiplication contractions similar to the Coppersmith-Winograd contractions, which can be expressed in the form $n^2.376$.

1 Introduction

The product of matrices is one of the most basic operations in mathematics and computer science. Many algorithms and data structures rely on the ability to multiply matrices efficiently. For instance, many computational problems, such as the fast Fourier transform, rely on matrix multiplication. The complexity of matrix multiplication has been a subject of extensive research, with the best-known algorithm for multiplying two $n \times n$ matrices using $O(n^{2.376})$ operations.

Until the 1960s, it was believed that the complexity of matrix multiplication was $O(n^3)$. In 1969, Volker Strassen proved that matrix multiplication could be done faster using a divide-and-conquer strategy, resulting in $O(n^{2.81})$ complexity. In 1986, Coppersmith and Winograd introduced a new algorithm that further reduced the exponent, achieving $O(n^{2.376})$ complexity. Since then, several improvements have been made, but the exponent remains the same.

In 2019, Lin and Francis at Microsoft introduced a new group-theoretic framework for improving the matrix multiplication algorithm. In 2020, together with Bremner and Shpectorov, they defined a new approach that could potentially lead to further improvements in the complexity of matrix multiplication.