Program execution on input of size n

- How many steps/cycles a processor would need to do
- Faster computer, larger input?
- But bad algorithm on fast computer will be outcompeted by good algorithm on slow...
- How to relate algorithm execution to nr of steps on input of size n? \( f(n) \)
- e.g. \( f(n) = n + n(n-1)/2 + 17n + n\log(n) \)

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.

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

Big-Oh notation classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Informed</th>
<th>Intuition</th>
<th>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) \in \omega(g(n)) )</td>
<td>f is dominated by g</td>
<td>Strictly below</td>
<td>(&lt;)</td>
</tr>
<tr>
<td>( f(n) \in \Omega(g(n)) )</td>
<td>Bounded from above and below</td>
<td>“equal to”</td>
<td>(=)</td>
</tr>
<tr>
<td>( f(n) \in \Theta(g(n)) )</td>
<td>Bounded from below</td>
<td>Lower bound</td>
<td>(\leq)</td>
</tr>
<tr>
<td>( f(n) \in O(g(n)) )</td>
<td>f dominates g</td>
<td>Strictly above</td>
<td>(&gt;)</td>
</tr>
<tr>
<td>( f(n) \in \Omega(g(n)) )</td>
<td>Bounded from above</td>
<td>Upper bound</td>
<td>(\geq)</td>
</tr>
</tbody>
</table>
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.

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
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
float RSums(float *a, const int n)
{
if (n <= 0)
    return 0;
else
    return (RSums(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
• usually input size 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: $F_n \approx 2^{0.618n}$.
(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 $Fib(n)$ 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 Fib2(n)
2. if n = 0 then return 0
3. else
4. introduce auxiliary array $F[0 \ldots n]$
5. $F[0] = 0$
6. $F[1] = 1$
7. for i = 2 to n do
9. end
10. return $F[n]$

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 \ldots n])
2. for $j = 2 \to n$ do
3. $a = A[j] \ ; \ j = j − 1$
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 $T_{A,i}$ the complexity of a single execution of lines $k$ thru $i$. Then:

- $T_{A,2}(n, i) \leq c_i$
- $T_{A,3}(n, i) \leq c_i + (i − 1)c_i$
- $T_{A,4}(n, i) \leq c_i + \sum_{j=1}^{i} (c_i + (i − j)c_i)$

Thus $T(n) = T_{A,7}(n) = O(n^2)$. 

The reason for the inefficiency is that the algorithm recomputes most of the values several (in fact, exponentially many) times:
Basic analysis rules

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

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

In analyzing 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?

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

$n = 1000$
\[ n = 10,000,000 \]

\[ n = 2,000,000,000 \]
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:** $2n^2 = O(n^3)$ ($c = 1, n_0 = 2$)

O-notation (lower bounds)

$\Omega$-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:** $\sqrt{n} = \Omega(\lg n)$ ($c = 1, n_0 = 16$)
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.

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

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

**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 \( f(n) = a_n n^m + \ldots + a_1 n + a_0 \), then \( f(n) = \Theta(n^m) \).

**Proof:**

\[
\begin{align*}
f(n) & = \sum_{i=m}^{n} a_i n^i \\
& \leq \sum_{i=m}^{\infty} |a_i| n^i \\
& \leq \sum_{i=m}^{\infty} |a_i| n^m, \text{ for } n \geq 1.
\end{align*}
\]

Therefore, let \( c = \sum_{i=m}^{\infty} |a_i| n_i = 1 \), we have
\[
f(n) \leq cn^m, \text{ for } n \geq n_c. \text{ Then, } f(n) = \Theta(n^m).
\]

---

**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** \( \to \infty \) as \( n \to \infty \)
- **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 \( f(n) \in O(g(n)) \)

**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:

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

Asymptotic Analysis

• Graphically, we can summarize these as follows:

We say

\[ f(n) = \begin{cases} 
O(g(n)) & \Theta(g(n)) \\
\omega(g(n)) & \Omega(g(n)) 
\end{cases} \]

if

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} 
0 & 0 < c < \infty 
\end{cases} \]

Asymptotic Analysis

• All of

\[ n^2 \quad 1000000 \quad n^2 - 4n + 19 \quad n^2 + 1000000 \]
\[ 323n^2 - 4n \ln(n) + 43n + 10 \quad 42n^2 + 32 \]
\[ n^2 + 61n \ln^2(n) + 7n + 14 \ln(n) + \ln(n) \]

are \Theta\ of each other

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

Asymptotic Analysis

• We will focus on these

\[ \Theta(1) \text{ constant} \]
\[ \Theta(\ln(n)) \text{ logarithmic} \]
\[ \Theta(n) \text{ linear} \]
\[ \Theta(n \ln(n)) \text{ “n-log-n”} \]
\[ \Theta(n^2) \text{ quadratic} \]
\[ \Theta(n^3) \text{ cubic} \]
\[ 2^n, e^n, 4^n, \ldots \text{ exponential} \]
\[ O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) \]

Growth of functions

• See Chapter “Growth of Functions” (CLRS)

Logarithms

- Algebra cheat sheet: [link to Algebra Cheat Sheet]
- [link to en.wikipedia.org/wiki/Logarithm]

Logarithms and Log Properties

<table>
<thead>
<tr>
<th>Logarithm Properties</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log_b x = \frac{\log_k(x)}{\log_k(b)} )</td>
<td>where ( \log_k(x) ) is ( x &gt; 0 )</td>
</tr>
</tbody>
</table>
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>$\equiv$</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, eventually...</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\omega(f(n))$</td>
<td>Small Omicron; Small O; Small Oh</td>
<td>f is dominated by g asymptotically for every $k$, $k &gt; 0$.</td>
<td>[ f(n) = o(g(n)) ]</td>
<td></td>
</tr>
<tr>
<td>$\Omega(f(n))$</td>
<td>Big Omega</td>
<td>f is bounded below by g asymptotically for some positive $k_2$, $k_2 &gt; 0$.</td>
<td>[ f(n) \geq \Omega(g(n)) ]</td>
<td></td>
</tr>
<tr>
<td>$\Theta(f(n))$</td>
<td>Big Theta</td>
<td>f is bounded both above and below by g asymptotically for some positive $k_1$, $k_2 &gt; 0$.</td>
<td>[ f(n) = \Theta(g(n)) ]</td>
<td></td>
</tr>
</tbody>
</table>

Functional iteration

We can use the notation $F^n(x)$ to denote the function $F$ iterated $n$ times applied times to an initial value of $x$. Formally, let $F$ be a function over the reals. For nonnegative integers $n$, we recursively define $F^0(x) = x$, $F^{n+1}(x) = F(F^n(x))$. For example, if $F(x) = 2x$, then $F^n(x) = 2^n x$.

The iterated logarithm function

We can use the notation $\lg^n x$ a small “log star” of $x$ to denote the iterated logarithm, which is defined as follows. Let $\lg x$ be as defined above, with $\lg 1 = 0$. Because the logarithm of a composite number is the sum of the logarithms of its prime factors, we have \[ \lg(n) = \sum_{p^k | n} \lg(p) \] for some prime factorizations $n = \prod_{p^k | n} p^k$. Repeated logarithms can be computed recursively using the identities \[ \lg^{n+1}(x) = \frac{\lg(x)}{\lg(\lg(x))} \text{ if } \lg(\lg(x)) > 1 \] \[ \lg^n(x) = \begin{cases} x & \text{if } x \leq 1 \\ \lg(x) & \text{if } \lg(x) > 1 \end{cases} \] for $n > 0$.
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

Lihtne sorteerimine

\[
\text{for } i = 2 \ldots n \\
\text{for } j = i; j > 1; j-- \\
\quad \text{if } A[j] < A[j-1] \\
\quad \quad \text{then swap}(A[j], A[j-1]) \\
\quad \quad \text{else next } i
\]

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

\[
\text{Merge-Sort}(A, p, r) \\
\text{if } p < r \text{ then } q = (p + r)/2 \\
\quad \text{Merge-Sort}(A, p, q) \\
\quad \text{Merge-Sort}(A, q+1, r) \\
\quad \text{Merge}(A, p, q, r)
\]

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 $\leq x$ elements in upper subarray.

![Partitioning subarray diagram]

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

3. **Combine:** Trivial.

**Key:** Linear-time partitioning subroutine.

---

**Pseudocode for quicksort**

```plaintext
QUICKSORT(A, p, r)
if $p < r$
  then $q \leftarrow$ PARTITION($A$, $p$, $r$)
  QUICKSORT($A$, $p$, $q-1$)
  QUICKSORT($A$, $q+1$, $r$)

Initial call: QUICKSORT($A$, 1, $n$)
```

---

**Partitioning subroutine**

```plaintext
PARTITION($A$, $p$, $q$) $\Rightarrow A[p \ldots q]$

$i \leftarrow p$
for $j \leftarrow p + 1$ to $q$
  do if $A[j] \leq x$
    then $i \leftarrow i + 1$
    exchange $A[i] \leftrightarrow A[j]$
exchange $A[p] \leftrightarrow A[i]$
return $i$

Invariant:

```

---

**Wikipedia / “video”**

---

**Kui palju võtab sorteerimine aega?**

- Võrdlustel põhinev: $A[i] \leftrightarrow A[j]$
  - Ülempiir – praegu parim teadaolev algoritm
  - Kas saame hinnata ülempiiri?

- Aga kui palju kahendotsimine?
- Aga kuidas lisada elemente tabelisse?
- (kahend-otsimis) Puu?

---

**Breaking the Coppermuth-Winograd barrier**

$n^{2.376}$

---
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...