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 \( n \)?
  - e.g. \( f(n) = n + n(n-1)/2 + 17 n + n \log(n) \)

### 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 \Omega(g(n)) )</td>
<td>Bounded from above</td>
<td>&quot;equal to&quot;</td>
<td>( \geq )</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 \Theta(g(n)) )</td>
<td>Bounded from above and below</td>
<td>&quot;equal to&quot;</td>
<td>( \equiv )</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 o(g(n)) )</td>
<td>f is dominated by g</td>
<td>Strictly below</td>
<td>(&lt;)</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. Relating to, connected or concerned with, quality or qualities. Now usually in implied or expressed opposition to quantitative.

#### 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
Program time and space complexity

- Time: count number of elementary calculations/operations during program execution

- Space: count amount of memory (RAM, disk, tape, flash memory, …)
  - usually we do not difference 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 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

- 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, … is defined by the well-known recursive formula:

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

This sequence arises in a natural way in countless applications. The numbers grow at an exponential rate: \( F_n \approx \phi^n \), where \( \phi \) is the golden ratio.

(Exercise.)
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

```plaintext
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

Let $T_{\text{insert}}(n)$ be the complexity of a single execution of lines 6-7. Then:

- $T_0(n, i, j) \leq c_1$
- $T_k(x/n, i, j) \leq c_2 + (i - 1)c_3$
- $T_k(x/n, i) \leq c_4 + c_5(j + 1) + c_6(1 - 1)c_7$
- $T_k(x/n, i, j) \leq c_8 + c_9(i(j + 1)) + c_{10}(1 - 1)c_{11}$
- $T_k(x/n) \leq c_{12} n + c_{13} n(n - 1)$

Thus $T(n) = T_{\text{insert}}(n) = O(n^2)$. 

Basic analysis rules

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

- $T[x\rightarrow x] = \text{constant}$.
- $T[\text{read } x] = \text{constant}$.
- $T[S_1, S_2, \ldots, S_r] = T[S_1] + \ldots + T[S_r]$.
- $T[\text{if } P \text{ then } S_1 \text{ else } S_2] = \begin{cases} T[P] + T[S_1] & \text{if } P = \text{true} \\ T[P] + T[S_2] & \text{if } P = \text{false} \end{cases}$.
- $T[\text{while } P \text{ do } S] = T[P]\cdot \left(\left(\sum_{i=1}^{\infty} P_i \right) + T[P]\right)$.

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?
• 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**

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

**Example:** $2n^2 \in O(n^3)$

**$\Omega$-notation (lower bounds):**

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

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

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

\[ Θ(g(n)) = O(g(n)) \cap Ω(g(n)) \]

**Example:** \[ \frac{1}{2} n^3 - 2n = Θ(n^3) \]

### O-notation and ω-notation

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

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

**Example:** \[ 2n^3 = o(n^5) \quad (n_0 = 2/c) \]

### 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 \( h(n) \in O(n^2) \).

### 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.

### O-notation and ω-notation

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

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

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

### 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:

$$f(n) = \sum_{i=0}^{\infty} a_i n^i \leq \sum_{i=0}^{\infty} |a_i| n^i = n^n \sum_{k=0}^{\infty} |a_i| n \quad \text{for } n \geq 1.$$ 

Therefore, let $c = \sum_{i=0}^{\infty} |a_i| n = 1$, we have $f(n) \leq c n^n$, for $n \geq n$. Thus, $f(n) = O(n^n)$.

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
- To formally describe equivalent run times, we will say that $f(n) = \Theta(g(n))$ if
  $$0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

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

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

Asymptotic Analysis
- We are also interested if one algorithm runs either asymptotically slower or faster than another.
  $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

  if this is true, we will say that $f(n) = \Theta(g(n))$

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

• To summarize:

\[
\begin{align*}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &> 0, \\
0 &< \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty, \\
\lim_{n\to\infty} \frac{f(n)}{g(n)} &< \infty
\end{align*}
\]

Asymptotic Analysis

• We have one final case:

\[
\begin{align*}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &> 0, \\
0 &< \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty, \\
\lim_{n\to\infty} \frac{f(n)}{g(n)} &< \infty
\end{align*}
\]

Asymptotic Analysis

• Graphically, we can summarize these as follows:

We say \( f(n) = \Theta(g(n)) \) if
\[
\begin{align*}
\lim_{n\to\infty} \frac{f(n)}{g(n)} = \theta, \\
0 &< \theta < \infty
\end{align*}
\]

Asymptotic Analysis

• All of

\[
\begin{align*}
323 n^2 - 4 n \ln(n) + 43 n + 10 &\quad 42n^2 + 32 \\
n^2 + 61 n \ln^2(n) + 7n + 14 \ln^3(n) + \ln(n) &\quad \text{are big-\( \Theta \) of each other}
\end{align*}
\]

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(2^n) < O(n!)

Growth of functions

• See Chapter “Growth of Functions” (CLRS)

  - [Link](http://ant.pku.edu.cn/~course/cs101/resource/Intro2Algorithm/book6/chap02.htm)
Logarithms


Logarithms and Log Properties

<table>
<thead>
<tr>
<th>Definition</th>
<th>Logarithms Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = \log_b x$ is equivalent to $x = b^y$</td>
<td>$\log_b 1 = 0$</td>
</tr>
<tr>
<td>Example</td>
<td>$\log_2 125 = 3$ became $5^3 = 125$</td>
</tr>
<tr>
<td>Special Logarithms</td>
<td>$\ln x = \log_e x$</td>
</tr>
<tr>
<td>$\log x = \log_{10} x$</td>
<td>$\log x = \log_{10} x$</td>
</tr>
<tr>
<td>$\log_{10} x$ is $\ln$</td>
<td>$\log x = \log_{10} x$</td>
</tr>
</tbody>
</table>

$\log_b x = \frac{\log_b (x)}{\log_b (b)}$


$\log_n x = \log_2 x$ (binary logarithm),
$\ln x = \log_e x$ (natural logarithm),
$\lg x = (\log x)^k$ (exponentiation),
$\log \lg x = \log (\log x)$ (composition).
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>$f(n) = o(g(n))$</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>$\leq$</td>
</tr>
<tr>
<td>$f(n) = \Theta(g(n))$</td>
<td>Bounded from above and below</td>
<td>“equal to”</td>
<td>$= $</td>
</tr>
<tr>
<td>$f(n) = \Omega(g(n))$</td>
<td>Bounded from below</td>
<td>Lower bound</td>
<td>$\geq$</td>
</tr>
<tr>
<td>$f(n) = \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>$f(n) = O(g(n))$</td>
<td>Big Omicron; Big O; Big Oh</td>
<td>$f$ is bounded above by $g$ (up to constant factor)</td>
<td>asymptotically for some $k$ or $(Note that, since the beginning of the 20th century, papers in number theory have been increasingly and widely using this notation in the weaker sense that $f(n) = o(g(n))$ is false)</td>
<td></td>
</tr>
<tr>
<td>$f(n) = \Omega(g(n))$</td>
<td>Big Omega</td>
<td>$f$ is bounded below by $g$ (up to constant factor)</td>
<td>asymptotically for some positive $k_1$, $k_2$</td>
<td></td>
</tr>
<tr>
<td>$f(n) = \Theta(g(n))$</td>
<td>Big Theta</td>
<td>$f$ is bounded both above and below by $g$ asymptotically for some positive $k_1$, $k_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f(n) = \omega(g(n))$</td>
<td>Small Omicron; Small O; Small Oh</td>
<td>$f$ is dominated by $g$ asymptotically for every $k$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f(n) = \Omega(g(n))$</td>
<td>Small Omega</td>
<td>$f$ dominates $g$ asymptotically for every $k$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Functional iteration

We use the notation $P^{(n)}(k)$ to denote the function $k$ iterations applied times to an initial value of $k$.

Formally, let $f$ be a function over the real, for nonnegative integers $n$, we recursively define:

$$f^{(0)}(k) = k$$

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

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

### The iterated logarithm function

We use the notation $\log^* n$ (a small “log star” or “log star of”) to denote the iterated logarithm, which is defined as follows:

Let $\log^* n$ be as defined above, with $\log^* 1 = 0$. Because the logarithm of a nonpositive number is undefined, we assume that $n > 0$. The iterated logarithm function is defined as:

$$\log^* n = \max(1, \log^* (\log n))$$

The iterated logarithm function is a very slowly growing function:

$$\log^* 2 = 1$$
$$\log^* 4 = 2$$
$$\log^* 16 = 3$$
$$\log^* 1024 = 4$$

Since the number of atoms in the observable universe is estimated to be about $10^{80}$, which is much less than $2^{2^{2^{2^{2^{2^{2^{2^{2^{2}}}}}}}}}$, we rarely encounter an input n such that $\log^* n = 5$. 

14.2.2013
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

\[
\begin{align*}
\text{for } i &= 2 \ldots n \\
\text{for } j &= i \ldots j > 1 \\
\text{if } A[j] \leq A[j-1] \\
\text{then } &\text{swap}(A[j], A[j-1]) \\
\text{else } &\text{continue}
\end{align*}
\]

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

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

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 $\leq x$ and elements in upper subarray $\geq x$.

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)

$x \leftarrow A[p]$
$i \leftarrow p$
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:** $\leq x \leq x \geq x$

---

Kui palju võtab sorteerimine aega?

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

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