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>$f(n) = \Theta(g(n))$</td>
<td>Bounded from above and below</td>
<td>&quot;equal to&quot;</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>

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."

**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**.

Program time and space complexity

• Time: count nr of elementary calculations/operations during program execution
  
  — usually we do not differentiate between cache, RAM, ...
  
  — In practice, for example, random access on tape impossible

• Space: count amount of memory (RAM, disk, tape, flash memory, ...)
  
  — Usually we do not differentiate between cache, RAM, ...

• 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. \( 4 \times n + 3 \)

• 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:

\[
F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad \text{if } n \geq 2.
\]

This sequence arises in a natural way in countless applications. The numbers grow at an exponential rate: \( F_n \approx 2^{n/2} \phi^{n/2} \).

(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 \( F(n) \)
2. if \( n = 0 \) then return 0
3. if \( n = 1 \) then return 1
4. else return \( F(n-1) + F(n-2) \)

This is, however, not a good idea, because the computation of \( F(n) \) 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 \( F(n) \)
2. if \( n = 0 \) then return 0
3. else introduce auxiliary array \( F[0 \ldots n] \)
4. for \( i = 2 \) to \( n \) do
5. \( F[i] = F[i-1] + F[i-2] \)
6. end
7. 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 \( \text{insertSort}(A[1 \ldots n]) \)
2. for \( i = 2 \) to \( n \) do
3. \( a = A[i], \ j = i - 1 \)
4. while \( j > 0 \) and \( a < A[j] \) do
5. \( A[j+1] \leftarrow A[j], \ j = j - 1 \)
6. end
7. \( A[j+1] \leftarrow a \)
8. end

Analysis of insertion sort

Denote: \( T_{\text{is}}(n, i, j) \) the complexity of a single execution of lines \( k \) thru \( l \). Then:

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

Thus \( T(n) = O(n^2) \).
Basic analysis rules
Denote $T[P]$ as the time complexity of an algorithm segment $P$.

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

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?
0.00001*n
100 n log n

n = 10,000,000

0.00001*n
100 n 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

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

**$\Theta$-notation (tight bounds)**

$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

**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

**$O$-notation and $\omega$-notation**

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

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

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

**Example:** $2n^2 = o(n^3)$ ($n_0 = 2/c$)

**$\omega$-notation and $\omega$-notation**

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

**Example:** $\sqrt{n} = \omega(\log n)$ ($n_0 = 1 + 1/c$)

**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 $h(n) \in O(n^2)$.
Theorem 1.2
If \( f(n) = a_n n^r + \ldots + a_i n^i + a_j \), then \( f(n) = O(n^r) \).

Proof:

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

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

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

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

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

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

- To formally describe equivalent run times, we will say that

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

- Note: this is not inequality – it would have been better if it said

\[
f(n) \equiv \Theta(g(n)) \text{ however, someone picked –}
\]

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

• To summarize:

\[
\begin{align*}
\lim_{n \to \infty} \frac{f(n)}{g(n)} &> 0 & \text{if } f(n) = \Omega(g(n)) \\
0 &< \lim_{n \to \infty} \frac{f(n)}{g(n)} & \text{if } f(n) = \Theta(g(n)) \\
\lim_{n \to \infty} \frac{f(n)}{g(n)} &< \infty & \text{if } f(n) = O(g(n))
\end{align*}
\]

Asymptotic Analysis

• We have one final case:

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

Asymptotic Analysis

• Graphically, we can summarize these as follows:

We say \( f(n) = \frac{O(g(n))}{o(g(n))} \) if

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

Asymptotic Analysis

• All of

\[
\begin{align*}
n^2 &< 100000 & n^2 - 4n + 19 &< n^2 + 1000000 \\
323n^2 - 4n \ln(n) + 43n + 10 &< 42n^2 + 32 \\
n^2 + 61n \ln^2(n) + 7n + 14 \ln^3(n) + \ln(n) &< n^2 + 32n^2 - 4n \ln(n) + 43n + 10
\end{align*}
\]

are big-\( \Theta \) of each other

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

Asymptotic Analysis

• We will focus on these

\[
\begin{align*}
\Theta(1) &\quad \text{constant} \\
\Theta(\ln(n)) &\quad \text{logarithmic} \\
\Theta(n) &\quad \text{linear} \\
\Theta(n \ln(n)) &\quad "n-\log-n" \\
\Theta(n^2) &\quad \text{quadratic} \\
\Theta(n^3) &\quad \text{cubic} \\
2^n, e^n, 4^n, ... &\quad \text{exponential}
\end{align*}
\]

Growth of functions

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


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

**Logarithms and Log Properties**

**Definition**

\( y = \log_b x \) is equivalent to \( x = b^y \)

**Logarithm Properties**

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

**Example**

\( \log_2 128 = 7 \) because \( 2^7 = 128 \)

**Special Logarithms**

\( \ln x = \log_e x \) natural log

\( \log_{10} x \) common log

where \( e \approx 2.718281828 \ldots \)

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

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

$$\log_b x = \Theta(\log_a x)$$

### Big-Oh notation classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Informal</th>
<th>Intuition</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(n) = \Theta(g(n))$</td>
<td>$f$ is dominated by $g$</td>
<td>&quot;equal to&quot;</td>
<td>$\approx$</td>
</tr>
<tr>
<td>$f(n) = \Omega(g(n))$</td>
<td>$f$ dominates $g$</td>
<td>Strictly above</td>
<td>$\geq$</td>
</tr>
<tr>
<td>$f(n) = O(g(n))$</td>
<td>$f$ is bounded above $g$</td>
<td>Upper bound</td>
<td>$\leq$</td>
</tr>
<tr>
<td>$f(n) = o(g(n))$</td>
<td>$f$ is dominated by $g$</td>
<td>Strictly below</td>
<td>$\leq$</td>
</tr>
</tbody>
</table>

### Family of Bachmann–Landau notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Name</th>
<th>Intuition</th>
<th>As, eventual...</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(n) = O(g(n))$</td>
<td>Big O; Big Oh</td>
<td>$f$ is bounded above by $g$ (up to constant factor)</td>
<td>asymptotically for some $k$ or $\varepsilon$</td>
<td>$f(n) \leq c \cdot g(n)$ for sufficiently large $n$</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 $k$</td>
<td>$f(n) \geq c \cdot g(n)$ for sufficiently large $n$</td>
</tr>
<tr>
<td>$f(n) = \Theta(g(n))$</td>
<td>Big Theta</td>
<td>$f$ is bounded between $g$ and $h$ asymptotically</td>
<td>for some positive $k_1$, $k_2$</td>
<td>$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for sufficiently large $n$</td>
</tr>
<tr>
<td>$f(n) = o(g(n))$</td>
<td>Small O; Small Oh</td>
<td>$f$ is dominated by $g$</td>
<td>asymptotically for every $\varepsilon$</td>
<td>$\frac{f(n)}{g(n)} \to 0$ as $n \to \infty$</td>
</tr>
<tr>
<td>$f(n) = \omega(g(n))$</td>
<td>Small Omega</td>
<td>$f$ dominates $g$</td>
<td>asymptotically</td>
<td>$\frac{f(n)}{g(n)} \to \infty$ as $n \to \infty$</td>
</tr>
</tbody>
</table>

### Functional Iteration

We use the notation $f^{(i)}(n)$ to denote the function $f(n)$ iterated $i$ times. For a nonnegative integer $n$, we recursively define $f^{(0)}(n) = n$. For example, $f^{(2)}(n) = 2n$, if $f(n) = 2n$.

The iterated logarithm function

We use the notation $\log^{(i)} n$ (read "log tower of $i$") to denote the iterated logarithm, which is defined as follows.

Let $\log^0 n = n$ be as defined above, with $\log^1 n = \log n$. Because the logarithm of a positive number is undefined, $\log^0 n$ is defined only for $n > 1$. To be rigorous, the logarithm function applied in succession, starting with argument $i$ from $\log^i n$, the logarithm of $n$ raised to the $i$th power. The iterated logarithm is defined as $\log^{(i)} n = \max (i \geq 0: \log^{(i)} n > 1)$.
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--
            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

```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.

Example

- Applying the merge sort algorithm:
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...
Breaking the Coppersmith-Winograd barrier

Virginia Vassilevska Williams
UC Berkeley and Stanford University

Abstract

No linear time factorized matrix multiplication is known in the Coppersmith-Winograd framework, and other factorized methods have a $\Omega(n^{2.688})$ bound.

I Introduction

The product of two matrices is one of the most basic operations in mathematics and computer science. Many important subroutines depend on efficient matrix multiplication. It is well known that the complexity of matrix multiplication is $O(n^3)$, i.e., the product of two $n \times n$ matrices can be computed in $O(n^3)$ time. However, there are many computational problems that, at the heart of their solving, are related to matrix multiplication.

Under the assumption that the complexity of matrix multiplication is $\Omega(n^{2.5})$ under worst-case complexity, Williams [W20] proved that the existence of a subcubic algorithm for matrix multiplication is equivalent to the existence of a subcubic algorithm for integer multiplication. This result is based on the assumption that matrix multiplication is a natural and relatively easy algorithm for matrix multiplication, and that this assumption is true. This assumption is also true for the case of integer multiplication, and it is also true for the case of polynomial multiplication.

In 2001, Coppersmith and Winograd [CW90] introduced a new group-theoretic framework for integer and matrix multiplication, which is based on the use of non-Abelian groups. These groups are defined by their multiplication tables, and they are used to construct efficient algorithms for matrix multiplication.

In 2003, Cohn and Franks [CF03] introduced a new group-theoretic framework for integer multiplication, which is based on the use of non-Abelian groups. These groups are defined by their multiplication tables, and they are used to construct efficient algorithms for integer multiplication.

Williams [W20] proved that the existence of a subcubic algorithm for integer multiplication is equivalent to the existence of a subcubic algorithm for matrix multiplication. This result is based on the assumption that matrix multiplication is a natural and relatively easy algorithm for matrix multiplication, and that this assumption is true. This assumption is also true for the case of integer multiplication, and it is also true for the case of polynomial multiplication.

In 2005, Cohn and Franks [CF03] introduced a new group-theoretic framework for integer multiplication, which is based on the use of non-Abelian groups. These groups are defined by their multiplication tables, and they are used to construct efficient algorithms for integer multiplication.

In 2007, Cohn and Franks [CF07] introduced a new group-theoretic framework for integer multiplication, which is based on the use of non-Abelian groups. These groups are defined by their multiplication tables, and they are used to construct efficient algorithms for integer multiplication.