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>$f(n) \in O(g(n))$</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>$\geq$</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>$f(n) \in \omega(g(n))$</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.

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 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 $F_0, F_1, ...$ 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: $F_n = 2^{\theta(n)}$.

(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
4. introduce auxiliary array \( F[0 \ldots n] \)
5. \( F[0] \leftarrow 0 \)
6. \( F[1] \leftarrow 1 \)
7. for \( i \leftarrow 2 \) to \( n \) do
8. \( F[i] \leftarrow F[i-1] + F[i-2] \)
9. end
10. return \( F[n] \)
11. 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 INSERT-SORT \( (A[1 \ldots n]) \)
2. for \( i \leftarrow 2 \) to \( n \) do
3. \( a \leftarrow A[j] ; j \leftarrow i - 1 \)
4. while \( j > 0 \) and \( a < A[j] \) do
5. \( A[j+1] \leftarrow A[j] ; j \leftarrow j - 1 \)
6. end
7. \( A[j+1] \leftarrow a \)
8. end

### Analysis of insertion sort

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

\[
T_{i,j} \leq c_1
\]

\[
T_{i,i} \leq c_2 + (i-1)c_1
\]

\[
T_{i,i} \leq c_2 + c_2 + (i-1)c_1
\]

\[
T_{i,i} \leq c_2 + \sum_{k=2}^{i} (c_2 + (i-1)c_1)
\]

\[
= c_2 + (i-1)(c_2 + c_2) + c_1 \sum_{k=2}^{i} (i-k)
\]

\[
\leq \text{const} \cdot n + c_1 \cdot \frac{1}{2} n(n-1)
\]

Thus \( T(n) = T_{i,i}(n) = O(n^2) \).
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^2, 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**: $2n^2 = O(n^3)$ ($c = 1$, $n_0 = 2$)

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

---

**o-notation and \( \omega \)-notation**

\( O \)-notation and \( \Omega \)-notation are like \( \leq \) and \( \geq \).

\( o \)-notation and \( \omega \)-notation are like \( < \) and \( > \).

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

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

---

**o-notation and \( \omega \)-notation**

\( O \)-notation and \( \Omega \)-notation are like \( \leq \) and \( \geq \).

\( o \)-notation and \( \omega \)-notation are like \( < \) and \( > \).

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

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

---

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

Proof:
\[
f(n) = \sum_{i=0}^{n} a_i n^i \leq \sum_{i=0}^{n} |a_i| n^i \\
\leq n^k \sum_{i=0}^{n} |a_i| n^i \\
\leq n^k \sum_{i=0}^{n} |a_i|, \text{ for } n \geq 1.
\]

Therefore, let \( c = \sum_{i=0}^{n} |a_i| \theta, \) we have
\[
f(n) \leq cn^k, \text{ for } n \geq \theta. \text{ Thus, } f(n) = O(n^k).
\]

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:
\[
\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}
\]

Asymptotic Analysis

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

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

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

• 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

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

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

Asymptotic Analysis

• If the limit is zero, i.e.,
\[
\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0
\]

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:

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

Asymptotic Analysis

• We have one final case:

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

Asymptotic Analysis

• Graphically, we can summarize these as follows:

\[
\begin{align*}
\text{We say} \quad f(n) = \begin{cases} 
O(g(n)) & \text{if } \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \\
o(g(n)) & \text{if } \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \\
\Theta(g(n)) & \text{if } 0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty \\
o(1) & \text{if } \lim_{n \to \infty} \frac{f(n)}{g(n)} < 0
\end{cases}
\end{align*}
\]

Asymptotic Analysis

• All of

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

are \( \Theta \) of each other

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

Asymptotic Analysis

• We will focus on these

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

Growth of functions

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

- Algebra cheat sheet: [Link](http://tutorial.math.lamar.edu/pdf/Algebra_Cheat_Sheet.pdf)

\[
\begin{align*}
\ln n &= \log_2 n \quad \text{(binary logarithm)}, \\
\ln n &= \log_2 n \quad \text{(natural logarithm)}, \\
\lg^n n &= (\ln n)^k \quad \text{(exponentiation)}, \\
\lg \lg n &= \lg(\ln n) \quad \text{(composition)}. \\
\end{align*}
\]
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>$f(n) \in \Omega(g(n))$</td>
<td>Bounded from below</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>&quot;equal to&quot;</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, eventually...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td>Big O; Big Oh</td>
<td>$f$ is bounded above by $g$ (up to constant factor)</td>
<td>$\leq$</td>
</tr>
<tr>
<td>$\Omega(n)$</td>
<td>Big Omega</td>
<td>$f$ is bounded below by $g$ (up to constant factor)</td>
<td>$\geq$</td>
</tr>
<tr>
<td>$\Theta(n)$</td>
<td>Big Theta</td>
<td>$f$ is bounded both above and below by $g$ (up to constant factor)</td>
<td>$\leq$ and $\geq$</td>
</tr>
<tr>
<td>$o(n)$</td>
<td>Small O; Small Oh</td>
<td>$f$ is dominated by $g$ asymptotically</td>
<td>$&lt;$</td>
</tr>
<tr>
<td>$\omega(n)$</td>
<td>Small Omega</td>
<td>$f$ dominates $g$ asymptotically</td>
<td>$&gt;$</td>
</tr>
</tbody>
</table>

Problems 3-2: Ordering by asymptotic growth rates

a. Rank the following functions by order of growth; that is, find an arrangement $p_1, p_2, ..., p_9$ of the functions satisfying $p_1 = o(p_2), p_2 = o(p_3), ..., p_8 = o(p_9)$. Partition your list into equivalence classes such that $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = O(g(n))$.

1. $\lg(n^2)$
2. $2^{\sqrt{n}}$
3. $\sqrt{\sqrt{n}}$
4. $n^2$
5. $n!$
6. $n! n^{\sqrt{n}}$
7. $\lg(n!)$
8. $\sqrt{\sqrt{n}}$
9. $\sqrt{\sqrt{n}}$
10. $\sqrt{\sqrt{n}}$
11. $\sqrt{\sqrt{n}}$
12. $\sqrt{\sqrt{n}}$
13. $\sqrt{\sqrt{n}}$
14. $\sqrt{\sqrt{n}}$
15. $\sqrt{\sqrt{n}}$
16. $\sqrt{\sqrt{n}}$
17. $\sqrt{\sqrt{n}}$
18. $\sqrt{\sqrt{n}}$
19. $\sqrt{\sqrt{n}}$
20. $\sqrt{\sqrt{n}}$
21. $\sqrt{\sqrt{n}}$
22. $\sqrt{\sqrt{n}}$
23. $\sqrt{\sqrt{n}}$
24. $\sqrt{\sqrt{n}}$
25. $\sqrt{\sqrt{n}}$
26. $\sqrt{\sqrt{n}}$
27. $\sqrt{\sqrt{n}}$
28. $\sqrt{\sqrt{n}}$
29. $\sqrt{\sqrt{n}}$
30. $\sqrt{\sqrt{n}}$
31. $\sqrt{\sqrt{n}}$
32. $\sqrt{\sqrt{n}}$
33. $\sqrt{\sqrt{n}}$
34. $\sqrt{\sqrt{n}}$
35. $\sqrt{\sqrt{n}}$
36. $\sqrt{\sqrt{n}}$
37. $\sqrt{\sqrt{n}}$
38. $\sqrt{\sqrt{n}}$
39. $\sqrt{\sqrt{n}}$
40. $\sqrt{\sqrt{n}}$
41. $\sqrt{\sqrt{n}}$
42. $\sqrt{\sqrt{n}}$
43. $\sqrt{\sqrt{n}}$
44. $\sqrt{\sqrt{n}}$
45. $\sqrt{\sqrt{n}}$
46. $\sqrt{\sqrt{n}}$
47. $\sqrt{\sqrt{n}}$
48. $\sqrt{\sqrt{n}}$
49. $\sqrt{\sqrt{n}}$
50. $\sqrt{\sqrt{n}}$
51. $\sqrt{\sqrt{n}}$
52. $\sqrt{\sqrt{n}}$
53. $\sqrt{\sqrt{n}}$
54. $\sqrt{\sqrt{n}}$
55. $\sqrt{\sqrt{n}}$
56. $\sqrt{\sqrt{n}}$
57. $\sqrt{\sqrt{n}}$
58. $\sqrt{\sqrt{n}}$
59. $\sqrt{\sqrt{n}}$
60. $\sqrt{\sqrt{n}}$
61. $\sqrt{\sqrt{n}}$
62. $\sqrt{\sqrt{n}}$
63. $\sqrt{\sqrt{n}}$
64. $\sqrt{\sqrt{n}}$
65. $\sqrt{\sqrt{n}}$
66. $\sqrt{\sqrt{n}}$
67. $\sqrt{\sqrt{n}}$
68. $\sqrt{\sqrt{n}}$
69. $\sqrt{\sqrt{n}}$
70. $\sqrt{\sqrt{n}}$
71. $\sqrt{\sqrt{n}}$
72. $\sqrt{\sqrt{n}}$
73. $\sqrt{\sqrt{n}}$
74. $\sqrt{\sqrt{n}}$
75. $\sqrt{\sqrt{n}}$
76. $\sqrt{\sqrt{n}}$
77. $\sqrt{\sqrt{n}}$
78. $\sqrt{\sqrt{n}}$
79. $\sqrt{\sqrt{n}}$
80. $\sqrt{\sqrt{n}}$
81. $\sqrt{\sqrt{n}}$
82. $\sqrt{\sqrt{n}}$
83. $\sqrt{\sqrt{n}}$
84. $\sqrt{\sqrt{n}}$
85. $\sqrt{\sqrt{n}}$
86. $\sqrt{\sqrt{n}}$
87. $\sqrt{\sqrt{n}}$
88. $\sqrt{\sqrt{n}}$
89. $\sqrt{\sqrt{n}}$
90. $\sqrt{\sqrt{n}}$
91. $\sqrt{\sqrt{n}}$
92. $\sqrt{\sqrt{n}}$
93. $\sqrt{\sqrt{n}}$
94. $\sqrt{\sqrt{n}}$
95. $\sqrt{\sqrt{n}}$
96. $\sqrt{\sqrt{n}}$
97. $\sqrt{\sqrt{n}}$
98. $\sqrt{\sqrt{n}}$
99. $\sqrt{\sqrt{n}}$
100. $\sqrt{\sqrt{n}}$

The Iterated logarithm is a very slowly growing function:

- $\log^2 2 = 1$
- $\log^2 4 = 2$
- $\log^2 16 = 3$
- $\log^2 65536 = 4$
- $\log^2 2^{65536} = 5$

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 $\log^2 n = 5$. 
How much time does sorting take?

  - 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 )
```

Example

- Applying the merge sort algorithm:

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.

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 $q \leftarrow$ PARTITION($A, p, r$)
    QUICKSORT($A, p, q-1$)
    QUICKSORT($A, q+1, r$)

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

Partitioning subroutine

```
PARTITION($A, p, q$) $\triangleright A[p \ldots q]$
  $x \leftarrow A[p]$ $\triangleright$ pivot $= A[p]$
  $i \leftarrow p$
  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: $\begin{array}{cccc}
  \leq & x & \geq & ? \\
  p & i & j & q
  \end{array}$
```

Conclusions

- Algorithm complexity deals with the behavior in the *long-term*
  - worst case $\quad$ typical
  - average case $\quad$ quite hard
  - best case $\quad$ bogus, “cheating”

- In practice, long-term sometimes not necessary
  - E.g. for sorting 20 elements, you don’t need fancy algorithms…
1 Introduction

The product of two matrices is one of the most basic operations in linear algebra and computer science. Many algorithms exist for performing this operation, but most are either time or space inefficient. The Fast Fourier Transform (FFT) has proven to be among the most efficient algorithms for computing the product of two matrices. However, many real-world applications require the computation of matrix multiplication. In this section, we will discuss some recent advances in this area.

In the late 1950s, a new technique for multiplying matrices was introduced by Strassen. This technique, known as the Strassen algorithm, reduces the complexity of matrix multiplication from $O(n^3)$ to $O(n^{log_27})$. In 1969, Shoup introduced a new algorithm for multiplying matrices, which he called the Coppersmith-Winograd algorithm. This algorithm further reduces the complexity of matrix multiplication to $O(n^{2.376})$.

In 1985, Strassen and Winograd introduced a new algorithm for multiplying matrices, which they called the Strassen-Winograd algorithm. This algorithm further reduces the complexity of matrix multiplication to $O(n^{2.3727})$. These algorithms have been refined over the years, and the current best known algorithm for matrix multiplication is an extension of the Strassen-Winograd algorithm, which has a complexity of $O(n^{2.373})$. This algorithm has been shown to be faster than previous methods for large matrices.