Algorithmics (6EAP)
MTAT.03.238
courses.cs.ut.ee
Order of growth... maths
Jaak Vilo
2020 fall
Coronavirus: cs.ut.ee/reg

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 \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>“equal to”</td>
<td>( = )</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
  - 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

• 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 the nr of elementary 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:

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

This sequence arises in a natural way in countless applications. The numbers grow at an exponential rate: \( F_n \approx \varphi^n \sqrt{5} \) (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 Fib1(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 . . . 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. return F[n]
9. end
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 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
6. end
7. A[j + 1] = a
8. end

Analysis of insertion sort

Denote: $T_{A,i,j}$ the complexity of a single execution of lines $k$ thru $i$. Then:

$$T_{S}(n, i, j) \leq c_1$$
$$T_{A,i}(n, i) \leq c_2 + (i - 1)c_1$$
$$T_{S,i}(n, i) \leq c_3 + (n - 1)c_2 + (i - 1)c_1$$

Thus $T(n) = T_{S,i}(n) = O(n^2)$. 
• So, we may be able to count a nr of operations needed

• What do we do with this knowledge?
\[ 0.00001n^2 \quad 100n \log n \]

\[ n = 10,000,000 \]

\[ 0.00001n^2 \quad 100n \log n \]

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

- plot \([1:10] 0.01 \times x, 5 \times \log(x), x \times \log(x)/3\)
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 exists constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

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

functions, not values

funny, “one-way” equality

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

Ω-notation (lower bounds)

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

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

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

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

*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 \( \Omega \)-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:** \( 2n^2 = o(n^3) \) \( (n_0 = 2/c) \)

**O-notation and ω-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:** \( \sqrt{n} = o(\log n) \) \( (n_0 = 1+1/c) \)

---

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

---

**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:

\[
\sum_{i=0}^{n} a_i n^i \leq \sum_{i=0}^{n} |a_i| n^i
\leq n^k \sum_{i=0}^{n} |a_i| \quad \text{for } n \geq n_0.
\]

Therefore, let $c = \sum_{i=0}^{n} |a_i| n_i = 1$, we have $f(n) \leq cn^k$, for $n \geq n_0$. 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 $\to \infty$ 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 $f(n) = \Omega(g(n))$

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:
  \[ f(n) = \Omega(g(n)) \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} > 0 \]
  \[ f(n) = \Theta(g(n)) \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} \in (0, \infty) \]
  \[ f(n) = O(g(n)) \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty \]

Asymptotic Analysis

• We have one final case:
  \[ f(n) = o(g(n)) \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \]
  \[ f(n) = \omega(g(n)) \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \]

Asymptotic Analysis

• Graphically, we can summarize these as follows:
  We say \( f(n) = \Theta(g(n)) \) if
  \[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \]
  and
  \[ \lim_{n \to \infty} \frac{g(n)}{f(n)} = \infty \]

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

Growth of functions

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

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

\textbf{Logarithms and Log Properties}

**Definition**

\[ y = \log_b x \text{ is equivalent to } x = b^y \]

**Example**

\[ \log_2 128 = 7 \text{ because } 2^7 = 128 \]

**Special Logarithms**

\[ \ln x = \log_e x \quad \text{natural log} \]
\[ \log_{10} x = \log_{10} a \quad \text{common log} \]

where \( e \approx 2.718281828 \ldots \)

**Logarithms Properties**

\[
\begin{align*}
\log_a 1 &= 0 \\
\log_a a &= 1 \\
\log_a b^x &= x \log_a b \\
\log_a(bc) &= \log_a b + \log_a c \\
\log_a \left( \frac{b}{c} \right) &= \log_a b - \log_a c \\
\log_a b^x &= x \log_a b \\
\log_a b &= \frac{\log_c b}{\log_c a}
\end{align*}
\]

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>$f(n) \in \Theta(g(n))$</td>
<td>$f$ is bounded above and below by $g$ asymptotically for some $k_1, k_2 &gt; 0$</td>
<td>&quot;equal to&quot;</td>
<td>$=$</td>
</tr>
<tr>
<td>$f(n) \in O(g(n))$</td>
<td>$f$ is bounded above by $g$ asymptotically</td>
<td>Upper bound</td>
<td>$\leq$</td>
</tr>
<tr>
<td>$f(n) \in \Omega(g(n))$</td>
<td>$f$ is bounded below by $g$ asymptotically</td>
<td>Lower bound</td>
<td>$\geq$</td>
</tr>
<tr>
<td>$f(n) \in o(g(n))$</td>
<td>$f$ is strictly below $g$ asymptotically</td>
<td>Strictly below</td>
<td>$&lt;$</td>
</tr>
<tr>
<td>$f(n) \in \omega(g(n))$</td>
<td>$f$ dominates $g$ asymptotically</td>
<td>Strictly above</td>
<td>$&gt;$</td>
</tr>
</tbody>
</table>

### Functional iteration

We use the notation $f^{(k)}(x)$ to denote the function $f(x)$ iteratively applied $k$ times to an initial value of $x$. Formally, set $f^{(0)}(x) = x$. For non-negative integers $k$, we recursively define $f^{(k)}(x) = f^{(k-1)}(f(x))$ if $k > 0$, and $f^{(k)}(x) = x$ if $k = 0$. For example, if $f(x) = 2x$, then $f^{(2)}(x) = 2 \cdot 2x = 4x$.

The iterated logarithm function

We use the notation $\lg^k(n)$ to denote the iterated logarithm, which is defined as follows. Let $\lg(n)$ be as defined above, with $\lg^0(n) = n$. Because the logarithm of a nonnegative number is undefined, $\lg^0(n)$ is defined only for $n \geq 2$. To define $\lg^k(n)$ for the logarithm function applied $k$ times in succession, starting with argument $n$ from $\lg(n)$ (the logarithm of $n$ raised to the $k$th power). The iterated logarithm function is defined as $\lg^k(n) = \lg(n)$. For example, $\lg^2(n) = \lg(\lg(n))$.
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

```
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.**

Merge sort

```
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:
Pseudocode for quicksort

\[
\text{QUICKSORT}(A, p, r) \\
\text{if } p < r \\
\quad \text{then } q \leftarrow \text{PARTITION}(A, p, r) \\
\quad \text{QUICKSORT}(A, p, q-1) \\
\quad \text{QUICKSORT}(A, q+1, r) \\
\text{Initial call: quicksort}(A, 1, n)
\]

Partitioning subroutine

\[
\text{PARTITION}(A, p, q) \quad A[p \ldots q] \\
\text{x} \leftarrow A[p] \quad \text{pivot } = A[p] \\
i \leftarrow p \\
\text{for } j \leftarrow p + 1 \text{ to } q \\
\quad \text{do if } A[j] \leq x \\
\quad \text{then } i \leftarrow i + 1 \\
\quad \text{exchange } A[i] \leftrightarrow A[j] \\
\text{exchange } A[p] \leftrightarrow A[i] \\
\text{return } i
\]

Invariant:

```
\begin{array}{cccc}
\hline
p & i & j & q \\
\hline
\end{array}
```

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...
Breaking the Coppersmith-Winograd barrier
Virginia Naumova Williams
UC Berkeley and Indiana University

Abstract

The product of two matrices is one of the most basic operations in mathematics and computer science. Many other matrix-based operations can be efficiently solved if a fast matrix multiplication algorithm is available. However, the fastest known algorithms are only slightly faster than the obvious algorithm and the efficiency gap is relatively small. A major breakthrough was achieved in 1986 by Coppersmith and Winograd, who analyzed the complexity of multiplication of matrices of order $n$ and showed that $n^2.375$ operations are sufficient.

1 Introduction

The product of two matrices is one of the most basic operations in mathematics and computer science. Many other matrix-based operations can be efficiently solved if a fast matrix multiplication algorithm is available. However, the fastest known algorithms are only slightly faster than the obvious algorithm and the efficiency gap is relatively small. A major breakthrough was achieved in 1986 by Coppersmith and Winograd, who analyzed the complexity of multiplication of matrices of order $n$ and showed that $n^2.375$ operations are sufficient.

2.3727

n2.376