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) \leq \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 \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>( \ngeq )</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

- 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.

Examples

- A quantitative way to report a particular room temperature would be "the temperature in this room is 23 degrees Celsius."
- A qualitative way to report room temperature would be to say "this room is warmer than it is outside."
- A quantitative way to describe the tree would be to say "The tree is 30 feet tall."
- A qualitative way to describe a tree would be to say "the tree is taller than the building."

http://en.wikipedia.org/wiki/Qualitative_data
Program time and space complexity

- Time: count nr of elementary calculations/operations during program execution
- Space: count amount of memory (RAM, disk, tape, ...) usually no 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 \( S_{\text{sum}}(n) = 1 \).

• 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. \( 4n + 3 \)
- 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: \( F_n \approx 2^{n/1.618}. \) (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!)
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 \( x \).
- 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

```
function INSERTSORT(A[1,...,n])
  for i = 2 to n do
    a = A[1] - i - 1
    while j > 0 and a < A[j] do
      end
    A[i] = a
  end
```

Analysis of insertion sort

Denote: \( T_{k,j} \) is the complexity of a single execution of lines \( k \) thru \( j \), then:

- \( T_1(n, i, j) \leq c_1 \)
- \( T_2(x(n, i, j)) \leq c_2 (i-1) \alpha \)
- \( T_3(x(n, i, j)) \leq c_3 + q_2 + (i-1) \alpha \)
- \( T_{2,3}(x(n, i, j)) \leq c_4 + \sum_{d=2}^{n} c_2 q_2 + c_2 + (i-1) \alpha \)

Thus:

\[
T(n) = T_{2,3}(x(n)) = O(n^2).
\]

Basic analysis rules

Denote \( T[P] \) - the time complexity of an algorithm segment \( P \):

- \( T[x \leftarrow e] = \text{constant} \)
- \( T[\text{read } x] = \text{constant} \)
- \( T[\text{write } x] = \text{constant} \)
- \( T[S_1, S_2, \ldots, S_a] = T[S_1] + \ldots + T[S_a] \)
- \( T[i f P \text{ then } S_1 \text{ else } S_2] = \begin{cases} T[P] + T[S_1] & \text{if } P \text{ is true} \\ T[P] + T[S_2] & \text{if } P \text{ is false} \end{cases} \)

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 calculate

• What do we do with this knowledge?

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

\[ n = 1000 \]
```plaintext
• 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 $f(n) < g(n)$

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

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, $n_0 = 2$)

functions, not values

funny, "one-way" equality

$\Omega$-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, $n_0 = 16$)
**Θ-notation (tight bounds)**

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

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

**Θ, O, and Ω**

![Figure 2.1 Graphic examples of the Θ, O, and Ω notations.](image)

In each part, the value of \( n_0 \) shown is the minimum possible value; any greater value would also work.

---

**ο-notation and ω-notation**

Ο-notation and Ω-notation are like ≤ and ≥. ο-notation and ω-notation are like < and >.

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

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

---

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

Proof:
\[
 f(n) = \sum_{i=0}^{\infty} a_i n^i \leq \sum_{i=0}^{\infty} |a_i| n^i \\
 \leq \alpha n^i \sum_{n=1}^{\infty} |a_i|, \text{ for } n \geq 1.
\]
Therefore, let \( c = \sum_{i=0}^{\infty} |a_i| n_i = 1 \), we have
\[
 f(n) \leq \alpha n^i, \text{ for } n \geq n_i. \text{ Thus, } f(n) = \Theta(n^n).
\]
Asymptotic Analysis

- To summarize:

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

Asymptotic Analysis

- We have one final case:

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

Asymptotic Analysis

- Graphically, we can summarize these as follows:

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

Asymptotic Analysis

- All of

\[
\begin{align*}
n^2 & & 100000n^2 - 4n + 19 & & n^2 + 1000000 \\
323n^2 - 4n \ln(n) + 43n + 10 & & 42n^2 + 32 \\
n^2 + 61n \ln^2(n) + 7n + 14 \ln(n) + \ln(n) & & \text{are big-} \Theta \text{ of each other}
\end{align*}
\]

Asymptotic Analysis

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

Asymptotic Analysis

- We will focus on these

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

Growth of functions

- See Chapter “Growth of Functions” (CLRS)

10.2.2012

Logarithms


Logarithms and Log Properties

<table>
<thead>
<tr>
<th>Definition</th>
<th>Logarithmic 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: ( \log_{10} 125 ) became ( 5 )</td>
<td>( \log_b b = 1 )</td>
</tr>
<tr>
<td>Special Logarithms</td>
<td></td>
</tr>
<tr>
<td>( \ln x = \log_e x ) natural log</td>
<td>( \log_{10} xy = \log_{10} x + \log_{10} y )</td>
</tr>
<tr>
<td>( \log_{10} x ) common log</td>
<td>( \log_{10} \frac{x}{y} = \log_{10} x - \log_{10} y )</td>
</tr>
</tbody>
</table>

where \( e \approx 2.718281828 \ldots \)

The domain of \( \log_{10} x \) is \( x > 0 \)

\[
\log_b (x) = \frac{\log_{10} (x)}{\log_{10} (b)}
\]
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>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(n) = \Theta(g(n))$</td>
<td>$f$ is dominated by $g$</td>
<td>Upper bound from above and below</td>
<td>$\approx$</td>
</tr>
<tr>
<td>$f(n) = \Omega(g(n))$</td>
<td>$f$ dominates $g$</td>
<td>Lower bound from below</td>
<td>$\geq$</td>
</tr>
<tr>
<td>$f(n) = O(g(n))$</td>
<td>$f$ is bounded above by $g$</td>
<td>Strictly below</td>
<td>$&lt;$</td>
</tr>
<tr>
<td>$f(n) = \omega(g(n))$</td>
<td>$f$ is not $O(g)$</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>
</tr>
</thead>
<tbody>
<tr>
<td>$f(n) = \Theta(g(n))$</td>
<td>Big Theta</td>
<td>dominates $g$ asymptotically for some positive $k_1, k_2$</td>
</tr>
<tr>
<td>$f(n) = O(g(n))$</td>
<td>Big O</td>
<td>bounded from above by $g$ asymptotically ( k &gt; 0 )</td>
</tr>
<tr>
<td>$f(n) = \Omega(g(n))$</td>
<td>Big Omega</td>
<td>bounded from below by $g$ asymptotically ( k &gt; 0 )</td>
</tr>
<tr>
<td>$f(n) = o(g(n))$</td>
<td>Small O</td>
<td>dominated by $g$ for every constant ( \epsilon &gt; 0 )</td>
</tr>
<tr>
<td>$f(n) = \omega(g(n))$</td>
<td>Small Omega</td>
<td>dominates $g$ for every constant ( \epsilon &gt; 0 )</td>
</tr>
</tbody>
</table>

Functional iteration

We use the notation $F(n)$ to denote the function $f$ iterated applied times to an initial value of $n$. Formally, let $F$ be a function over the real numbers, the iterated function $F$ is recursively defined as:

$$F^{(n)}(x) = \begin{cases} \lim_{k \to \infty} F^{(k)}(x), & n \geq 0 \setminus \mathbb{N} \\ F^{(n)}(x), & n \in \mathbb{N} \end{cases}$$

For example, if $f(x) = x + 1$, then $F^{(n)}(x) = n - x$.

The iterated logarithm function

We use the notation $\lg^n x$ a small "log star of $x$" to denote the iterated logarithm, which is defined as follows. Let $\lg^n x$ be as defined above, with $\lg^n x = 0$ because the logarithm of a non-positive number is undefined. Then $\lg^{n+1} x$ is defined as either $\lg(\lg^n x)$ if $n > 0$, or $\lg^n x$ if $n = 0$. Since the number of stars in the observable universe is estimated to be about $10^{80}$, which is much less than $2^{2^{2^{2^{2^{65}}}}}$, we rarely encounter an input $x$ such that $\lg^n x = 0$. 

The iterated logarithm is a very slowly growing function.
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

```
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

```
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:

Wikipedia / viz.
Divide and conquer

Quick sort 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.
   Key: Linear-time partitioning subroutine.

Pseudocode for quicksort

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

Partitioning subroutine

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

Invariant:

\[
\begin{array}{ccccccc}
\_ & \_ & \_ & \_ & \_ & \_ & \_ \\
p & \_ & \_ & \_ & \_ & \_ & q
\end{array}
\]

Running time $= O(n)$ for $n$ elements.

Wikipedia / “video”

Kui palju võtab sorteirimine aega?

- Võrdlusest põhinev: $A[i] \leq A[j]$
  - Ülemist – praegu parim teadaolev algoritm
  - Kas saame hinnata alampiiri?

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

\[
\eta^2 = 3.727
\]

\[
\eta^2 = 3.766
\]
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...