Asymptotics

- Stating precise running times can be a daunting task, and may depend on implementation.
  - Ex: add two m by n matrices.

$$
\begin{array}{l}
\textbf{for } 1 \le i \le m \textbf{ do} \\
\quad \textbf{for all } 1 \le j \le n \textbf{ do} \\
\quad\quad C_{ij} = A_{ij} + B_{ij} \\
\quad \textbf{end for} \\
\textbf{end for}
\end{array}
$$

  - Intuitive running time: 2nm .... but:
    - how is the for loop implemented?
    - does C need to be initialized?
    - is it OK to assume that accessing C_{ij} is an atomic operation?

Time Complexity Function

Definition:

Given an algorithm A, a time complexity function is a function f of the input-size that whose value is an upper bound on the maximum number of steps that the algorithm performs.

Ex: The time complexity function for the matrix addition algorithm is $f(n,m)=cnm$.

We say that an algorithm has running time $f(n)$.

Asymptotics...

- Real running time of algorithm may be $\mathrm{c} \cdot nm$ for some c>2.
- Now assume that A_1 and A_2 are shortest path algorithms with running time $\mathrm{c}_1 \cdot n \log n$ and $\mathrm{c}_2 \cdot n^2$. **Which one is better?**
    - ➡ depends on c_1, c_2 and n.
- For large instances, however, algorithm A_1 is preferable, as there is n_0 such that $\mathrm{c}_1 \cdot n \log n \leq c_2 \cdot n^2$ for all $\mathrm{n} \geq n_0$

Big-"O" Notation

We develop algorithms for hard instances, and therefore care most about order of polynomial of running time!

Definition:

The running time g(n) of a given algorithm is O(f(n)) if there are c and $\mathrm{n_0}$ such that

$$\mathrm{g(n)} \leq c \cdot f(n)$$

for all $\mathrm{n} \geq n_0$.

We write: g(n) = O(f(n)).

"O"-Notation: Some Examples

- f(n)=$\sum_{i=0}^{d} c_i \cdot n^i = O(n^d)$
- $\mathrm{f}_1(n), \ldots, f_d(n) = O(h(n))$ and d is a constant, then $\mathrm{f}_1(n) + \ldots + f_d(n) = O(h(n))$
- $\mathrm{f(n)} = \mathrm{O(g(n))}$ and $\mathrm{g(n)} = \mathrm{O(h(n))}$, then $\mathrm{f(n)} = \mathrm{O(h(n))}$

Example: Matrix Addition

for $1 \leq i \leq m$ **do**
  for all $1 \leq j \leq n$ **do**
    $C_{ij} = A_{ij} + B_{ij}$
  end for
end for

➡ runs in time O(nm)

Caveat - Size of Numbers

- Assuming that arithmetic operations are constant-time is sometimes incorrect!
  - if numbers involved are large, computers may need several words for their storage
  - this leads to super-constant running times even for simple arithmetic operations!

- Assume: all graph attributes are bounded by polynomials in the input size!

Good Algorithms

- An algorithm is good if it runs fast!
  - more precisely: suppose we want to compare algorithms
    A and B with time complexity functions f and g, then we should prefer A if $f(n)=O(g(n))$.
- In general, call an algorithm efficient if its running time is a polynomial in the input size.
  - algorithms with exponential running times are inefficient!

Running Time Comparison

n \ f(n)	n	n $\log n$	n^2	n^3	1.5^n	2^n	n!
10							4 sec
30						18 min	10^{25} years
50		< 1 sec			11 min	36 years	
100				1 sec	12892 ye.	10^{17} years	
1000			1 sec	18 min			
10000			2 min	12 days		very long	
100000		2 sec	3 hours	32 years			
1 mio	1 sec	20 sec	12 days	31710 ye.			

Running times on a computer that executes 1 mio atomic operations per second.