Data Structures – LECTURE 2

Elements of complexity analysis

- Performance and efficiency
- Motivation: analysis of Insertion-Sort
- Asymptotic behavior and growth rates
- Time and space complexity
- Big-Oh functions: O(f(n)), Ω(f(n)), Θ(f(n))
- Properties of Big-Oh functions

Performance and efficiency

- We can quantify the <u>performance of a program</u> by measuring its run-time and memory usage.

  It depends on how fast the computer is, how good the compiler is, etc. → a very local and partial measure!
- We can quantify the <u>efficiency of an algorithm</u> by calculating its space and time requirements as a function of the basic units (memory cells and operations) it requires → independent of the implementation technology, but is only a guideline!

Example of a program analysis

Sort the array A of n integers

<u>Insertion-Sort(A)</u>	*cost*	*times*
1. for j = 2 to A.length	*c1*	*n*
2. do key ← A[j]	*c2*	*n –1*
3. // Insert A[j] into A[1..j-1]		
4. i ← j – 1	*c4*	*n –1*
5. while i > 0 and A[i] > key	*c5*	$\sum_{j=2}^{n} t_j$
6. do A[i+1] ← A[i]	*c6*	$\sum_{j=2}^{n} (t_j - 1)$
7. i ← i – 1	*c7*	$\sum_{j=2}^{n} (t_j - 1)$
8. A[i+1] ← key	*c8*	*n –1*

Insertion-Sort example

5 2 4 6 1 3

2 5 4 6 1 3

2 4 5 6 1 3

2 4 5 6 1 3

1 2 4 5 6 3

1 2 3 4 5 6

Program analysis method

- The running time is the sum of the running times of each statement executed
- Each statement takes c_i steps to execute and is executed t_i times → total running time is

$$T(n) = \sum_{i=1}^{k} c_i t_i$$

Insertion-Sort analysis (1)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- <u>Best case</u>: the array is in sorted order, so t_j=1 for j=2..n; step 5 takes $\sum_{j=2}^{n} t_j = n$ and steps 6 and 7 are not executed. Thus

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

$\boxed{T(n) = an + b}$ *This is a linear function!*

Insertion-Sort Analysis (2)

- Worst case: the array is in reversed sorted order, so $t_j = j$ for j=2..n, so step 5 takes

$$\sum_{j=2}^{n} j = n(n+1)/2 - 1$$

and steps 6 and 7 are always executed, so they take $\sum_{j=2}^{n} (j-1) = n(n+1)/2$

Overall

$$T(n) = (c_5 + c_6 + c_7)\frac{n^2}{2} + (c_1 + c_2 + c_4 + c_8 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2})n - (c_2 + c_4 + c_5 + c_8)$$

$$\boxed{T(n) = an^2 + bn + c \quad \textit{This is a quadratic function!}}$$

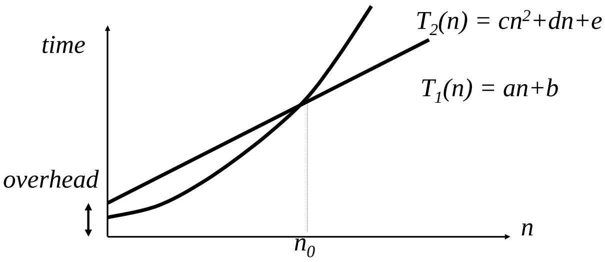
Asymptotic analysis

- We can write the running time of a program $T(n)$ as a function of the input size *n:*

  $T(n) = f(n)$
- The function contains constants that are program and platform-dependent.
- We are interested in the asymptotic behavior of the program: how quickly does the running time grow as a function of the input size?

  → Rationale: if the input size n is small, all programs are likely to do OK. But they will have trouble when n grows. In fact, the program performance will be dominated by it!

Asymptotic analysis: best vs. worst case for Insertion-Sort



No matter what the constants are $T_2(n) > T_1(n)$ after a while

To summarize

- The efficiency of an algorithm is best characterized by its asymptotic behavior as a function of the input or problem size n.
- We are interested in both the run-time and space requirements, as well as the best-case, worst-case, and average behavior of a program.
- We can compare algorithms based on their asymptotic behavior and select the one that is best suited for the task at hand.

Time and space complexity

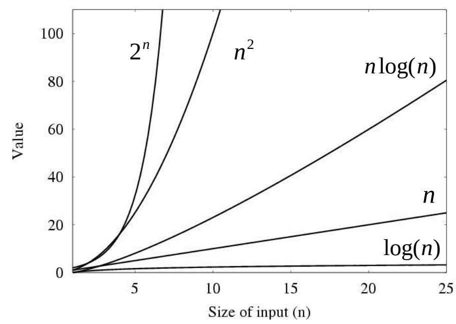
- Time complexity: $T(n)$

  How many operations are necessary to perform the computation as a function of the input size n.
- Space complexity: $S(n)$

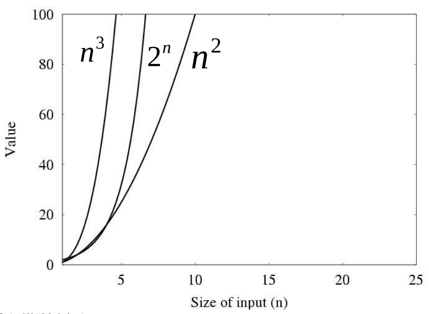
  How much memory is necessary to perform the computation as a function of the input size n.
- Rate of growth:

  We are interested in how fast the functions $T(n)$ and $S(n)$ grow as a function of the input size n.

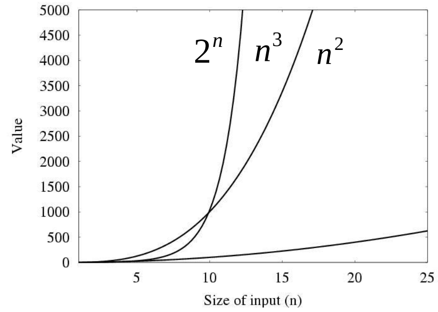
Rates of growth of common functions



Rates of growth: behavior (1)



Rates of growth: behavior (2)



Problem size as a function of time

Algorithm	**Time Complexity**	**Maximum problem size**		
		1 sec	**1 min**	**1 hour**
A_1	n	*1000*	*6 x10*^{4}	*3.6* x 10^6
A_2	$n \log_2 n$	*140*	*4893*	*2.0* x 10^5
A_3	n^2	*31*	*244*	*1897*
A_4	n^3	*10*	*39*	*153*
A_5	2^n	*9*	*15*	*21*

Assuming one unit of time equals one millisecond.

Effect of a tenfold speedup

Algorithm	**Time Complexity**	**Maximum problem size**	
		before speed-up	**after speed-up**
A_1	n	s_1	$10s_1$
A_2	$n \log_2 n$	s_2	*approx.* $10s_2$ *(for large* s_2*)*
A_3	n^2	s_3	$3.16s_3$
A_4	n^3	s_4	$2.15s_4$
A_5	2^n	s_5	$s_5 + 3.3$

Asymptotic functions

Define mathematical functions that estimate the complexity of algorithm A with a growth rate that is independent of the computer hardware and compiler. The functions ignore the constants and hold for sufficiently large input sizes n.

- **Upper bound** O(f(n)): at most f(n) operations
- **Lower bound** Ω(f(n)): at least f(n) operations
- **Tight bound** Θ(f(n)) : order of f(n) operations

Asymptotic upper bound – Big-Oh

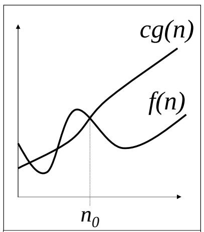
Let $f(n)$ and $g(n)$ be two functions from naturals to positive reals

$\boldsymbol{f(n) = O(g(n))}$

if there exist $c > 0$ and $n_0 > 1$ such that

$$f(n) \leq c(g(n))$$

for all $n \geq n_0$



f(n) =O(g(n))

Asymptotic lower bound – Big-Omega

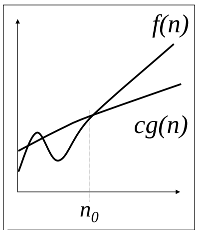
Let $f(n)$ and $g(n)$ be two functions from naturals to positive reals

$\boldsymbol{f(n) = \Omega(g(n))}$

if there exist $c > 0$ and $n_0 > 1$ such that

$$f(n) \geq c(g(n))$$

for all $n \geq n_0$



$\boldsymbol{f(n) = \Omega(g(n))}$

Asymptotic tight bound – Big-Theta

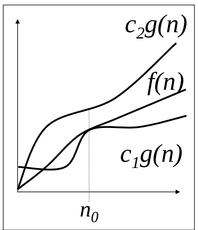
Let $f(n)$ and $g(n)$ be two functions from naturals to positive reals

$\boldsymbol{f(n) = \Theta(g(n))}$

if there exist $c_1, c_2 > 0$ and $n_0 > 1$ such that

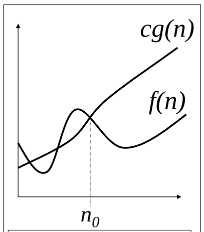
$$0 \leq c_1(g(n)) \leq f(n) \leq c_2(g(n))$$

for all $n \geq n_0$

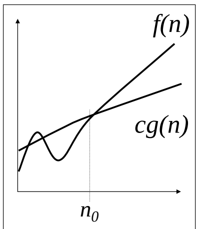


$f(n) = \Theta(g(n))$
Tight bound

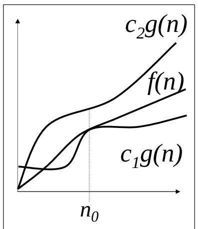
Graphs for O, Ω, and Θ



$f(n) = O(g(n))$
Upper bound



$f(n) = \Omega(g(n))$
Lower bound



$f(n) = \Theta(g(n))$
Tight bound

Properties of the O, Ω, and Θ functions

Theorem: f is tight iff it is an upper and a lower bound:

$f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

- Reflexivity:
  $f(n) = O(f(n))$; $f(n) = \Omega(f(n))$; $f(n) = \Theta(f(n))$
- Symmetry:
  $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$
- Transitivity:
  $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$
  $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ then $f(n) = \Omega(h(n))$
  $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ then $f(n) = \Theta(h(n))$

Properties of the O, Ω, and Θ functions

For O, Ω, and Θ:

- $O(O(f(n)) = O(f(n))$
- $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
- $O(f(n).g(n)) = O(f(n)).O(g(n))$
- $O(\log n) = O(\lg n)$ — lg is $\log_2$
- Polynomials: $O(\sum_{i=1}^{k} a_i n^i) = O(n^k)$
- Factorials:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

$O(n!) = O(n^{n+0.5})$
$O(\log n!) = O(n \lg n)$

Asymptotic functions

- Asymptotic functions are used in conjunction with recurrence equations to derive complexity bounds
- Proving a lower bound for an algorithm is usually harder than proving an upper bound for it. Proving a tight bound is hardest!
- Note: still does not answer the if this is the least or the most work for the given problem. For this, we need to consider upper and lower problem bounds (later in the course).