CPSC 335: Algorithm Engineering

Instructor: Dr. Doina Bein

- The complexity of some instances is less than or equal the worst case complexity
- Sometimes small instances do not follow the general trend
- Example: the *minimum selection problem*

  input: a list L of $n \geq 0$ numbers

  output: the minimum value among all elements in L
- Algorithm that solves the problem:

```
def naïve_min (n, L):
  if L is empty, return 0
  else
      Let min = first element of L (there is one since n > 0)
      For each element in L do
          if (min > element) then let min = element
      Return min
```

- Running time:

$$T = \begin{cases} 2 & if\ n = 0 \\ 2(n+1) & if\ n > 0 \end{cases}$$

- This value is a function of n, so we call it:

$$T(n) = \begin{cases} 2 & if\ n = 0 \\ 2(n+1) & if\ n > 0 \end{cases}$$

- Note: *When analyzing an algorithm, we assume that there is some threshold for the input size, beyond which the trend is established.*
- Thus we ignore small inputs from our analysis

- The measurement of any resource (time, space) involves a hidden constant:
- When analyzing the time complexity, we compute the number of steps performed by the algorithm on the RAM model
- Each simple instruction takes a precise amount of CPU time on a given computer, and a set of simple instructions (such as an algorithm) will take a multiplicative time
- Takeaway: From the point of algorithms' efficiency, two functions $T_1(n)$ and $T_2(n) = c \cdot T_1(n)$ that differ by only a constant are considered equivalent.

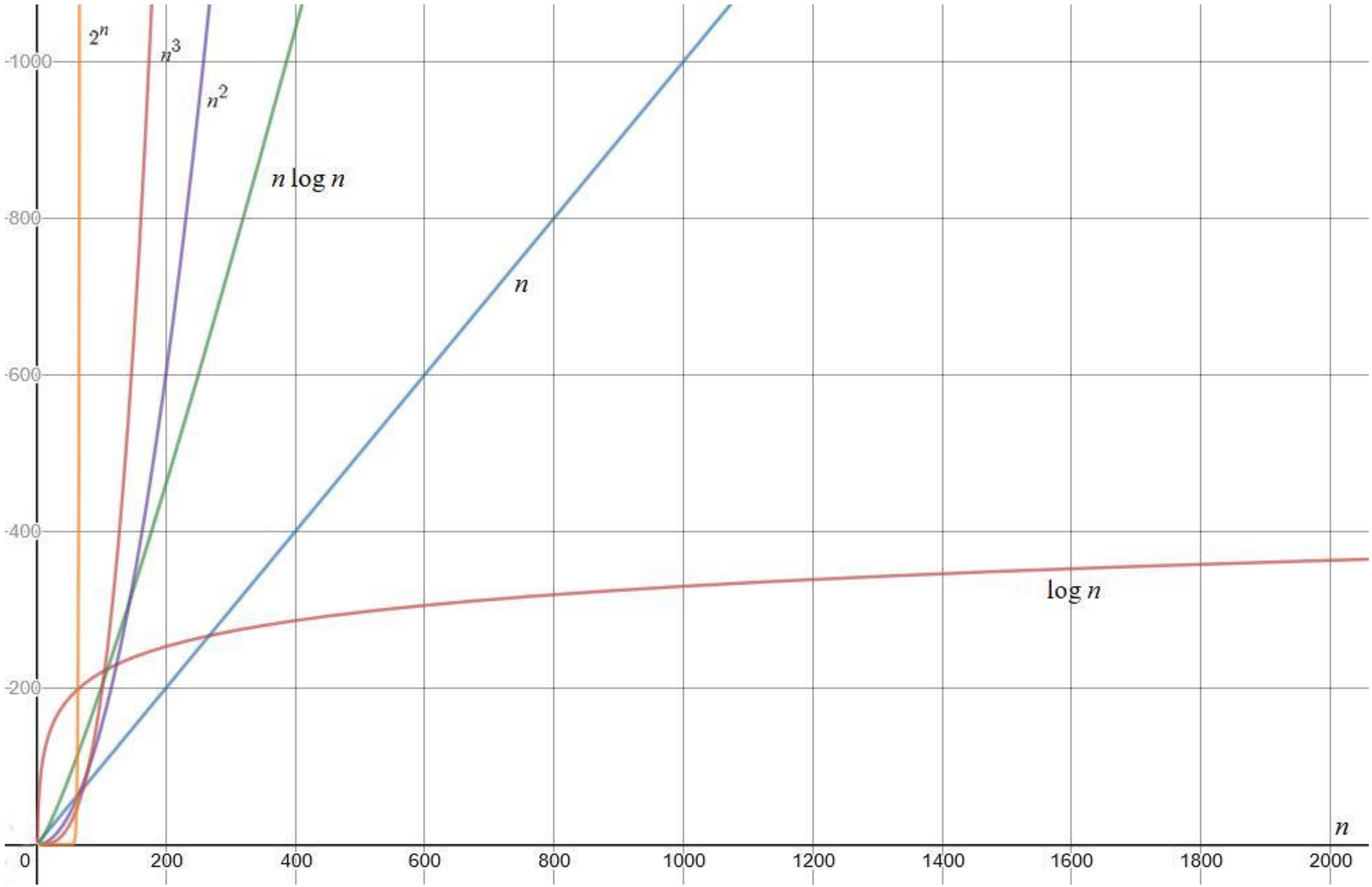
Order of Growth or Rate of Growth

- The time complexity of an algorithm is measured in terms of its *growth rate* (i.e. order) as a function of input size
- Sometimes computing the exact running time is not worth the effort
- Ex: minimum selection problem in a list of size n

```
def naïve_min (n, L):
  if L is empty, return 0
  else
        Let min = first element of L (there is one since n > 0)
        For each element in L do
            if (min > element) then let min = element
        Return min
```

- The execution time is proportional to n, the size of the input
- Hence the execution time of Algorithm naïve_min is linear i.e. $O(n)$ ("Big-Oh of n")

- We will define notation Big-Oh soon
- There are two simplifying assumptions that we make when analyzing the running time of an algorithm
    1. Only the leading term is considered
    2. Constants are ignored
- A solution that takes constant time has r.t. of O(1).
- An algorithm A is more efficient than another algorithm B if the w.c.r.t of A has a lower order of growth. Examples: n, $\log(n)$, n^2, $n^2\log(n)$, n^3, n^n



2^n
n^3
n^2
$n \log n$
n
$\log n$
1000
800
600
400
200
0
200
400
600
800
1000
1200
1400
1600
1800
2000
n

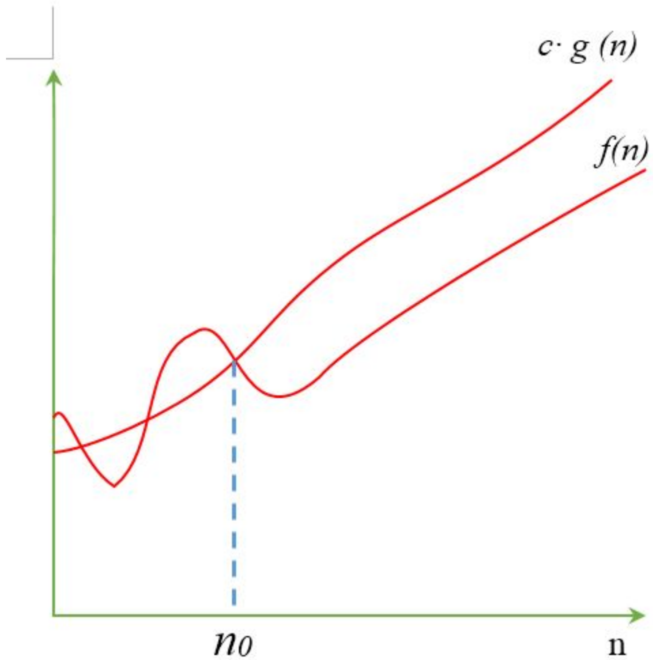
- There can be many functions between the consecutive functions shown
- $n^2 \leq n^2 \log \mathrm{n} \leq \ n^3$
- $\log n \leq \sqrt{n} \leq n$
- So on

Asymptotic Notation for Growth of Functions

- For large inputs, only the order of growth is relevant. We say then that we are studying the asymptotic efficiency of algorithms
- We are concerned with how the running time of an algorithm increases with the size of the input *in the limit* (i.e. the size of the input increases without bound)
- An algorithm that is asymptotically more efficient beats the rest for all but very small inputs

Big-Oh (O) Notation

- For a given function g(n), O(g(n)) denotes the set of functions f(n) (f(n) = O(g(n))) for which there exist two positive constants, c and n_0 such that f(n) ≤ c g(n) when n ≥ n_0.



The top efficiency classes

Notation	Name	Example
$O(1)$	constant	Evaluating one statement
$O(\log n)$	logarithmic	Searching a balanced search tree
$O(n)$	linear	For loop
$O(n \log n)$	"n-log-n"	Fast sorting algorithms such as heap sort or merge sort
$O(n^2)$	quadratic	Two nested for loops
$O(n^3)$	cubic	Three nested for loops
$O(c^n)$	exponential	All subsets of an n-element set
$O(n!)$	factorial	All permutations of an n-element sequence

- Example 1: $f_1(n) = 10n + 5$, $f_2(n) = n^2$. Then $f_1(n) = O\big(f_2(n)\big)$. Why?

$10n + 5 \leq c \cdot n^2$, $n > n_0$

Let us choose n_0=1 and c =15:

$10n + 5 \leq 15 \cdot n^2$, n >1

Or $10n + 5 \leq 10n^2 + 5n^2$, n > 1

This is trivially true.

n^2 is an upper bound of 10n+5

$f_2(n)$ is an upper bound of $f_1(n)$.

- Example 2: Show that $10n + 100 = O(n)$

Have $f(n) = 10n + 100$ and $g(n) = n$.

$10n + 100 \leq c \cdot n$, n > n_0

Let us choose n_0=1 and c =110:

$10n + 100 \leq 110 \cdot n$, n >1

$10n + 100 \leq 10n + 100n$, n >1

This is trivially true.

- Example 3: Show that $5n^3 + 100n\log n = O(n^3)$

$5n^3 + 100n\log n \leq 105n^3$, n > 1

or, $5n^3 + 100n\log n \leq 5n^3 + 100n^3$, n > 1

(This is trivially true.)

- Remark: $n^2 = O(n^3), n^2 = O(n^4), n^2 = O(n^3 \log n)$ and so on.

But $n^2 = O(n^2)$ is the tight upper bound

- Example 4: Prove that every polynomial $p(n) = a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \cdots + a_0$, with $a_k > 0$ belongs to $O(n^k)$.

Let SM be the sum of absolute values of all a_k, a_{k-1}, …, a_0.

Then we can write

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \cdots + a_0$$

$$\leq |a_k| n^k + |a_{k-1}| n^k + |a_{k-2}| n^k + \cdots + |a_m|$$

$$\leq (|a_k| + |a_{k-1}| + \cdots + |a_0|) n^k, \text{ n>1}$$

$$\leq SM \cdot n^k, \text{ n>1}$$

Thus $p(n) = O(n^k)$

- Example 5: Show that the statement $3n^2 + 2n + 10 = O(n)$ is false.
  We can also write that $3n^2 + 2n + 10 \notin O(n)$.

Assume $3n^2 + 2n + 10 = O(n)$. Thus there exist c and n_0 such that $3n^2 + 2n + 10 \leq c \cdot n$, $n > n_0$

Or, $3n + 2 + \frac{10}{n} \leq c$, $n > n_0$

Remember that c is a constant. Also $3n + 2 + \frac{10}{n}$ gets arbitrarily very large for large values of n, thus it cannot be upper bounded by a constant. Contradiction.

- Example 6: Let α and β be real positive numbers such that $0 < \alpha < \beta$. Show that n^{α} is in $O(n^{\beta})$.
- How to solve this problem?
- We use the following theorem:

Theorem: If T and f are univariate complexity functions, $f(n) > 0$, $\lim_{n \to \infty} \frac{T(n)}{f(n)} = L$, *and L is non-negative and constant with respect to n then* $T(n) = O(f(n))$.

Let $\beta - \alpha = \epsilon$, ϵ is positive.

We take to the limit $\lim_{n \to \infty} \frac{n^{\alpha}}{n^{\beta}} = \lim_{n \to \infty} \frac{n^{\alpha}}{n^{\alpha+\epsilon}} = \lim_{n \to \infty} \frac{1}{n^{\epsilon}} = 0.$

L=0 is a non-negative constant with respect to n thus $n^{\alpha} = O(n^{\beta})$.

Using limits, show that:

- Show that $10n + 5 = O(n^2)$
- Show that $10n + 100 = O(n)$
- Show that $5n^3 + 100n \log n = O(n^3)$
- Prove that every polynomial $p(n) = a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \cdots + a_0$, with $a_k > 0$ belongs to $O(n^k)$.
- $17 = O(18)$.

Remember that:

- We can drop additive constants
- We can drop multiplicative constants
- We can drop dominated terms (and keep only the dominating term); due to the

Lemma: For any complexity functions $f_0(n)$ *and* $f_1(n)$*,* $O\big(f_0(n) + f_1(n)\big) = O\Big(max\big(f_0(n), f_1(n)\big)\Big)$

Thus $O(n^2 + 2^n) = O(2^n)$

- We can drop floor and ceiling operators

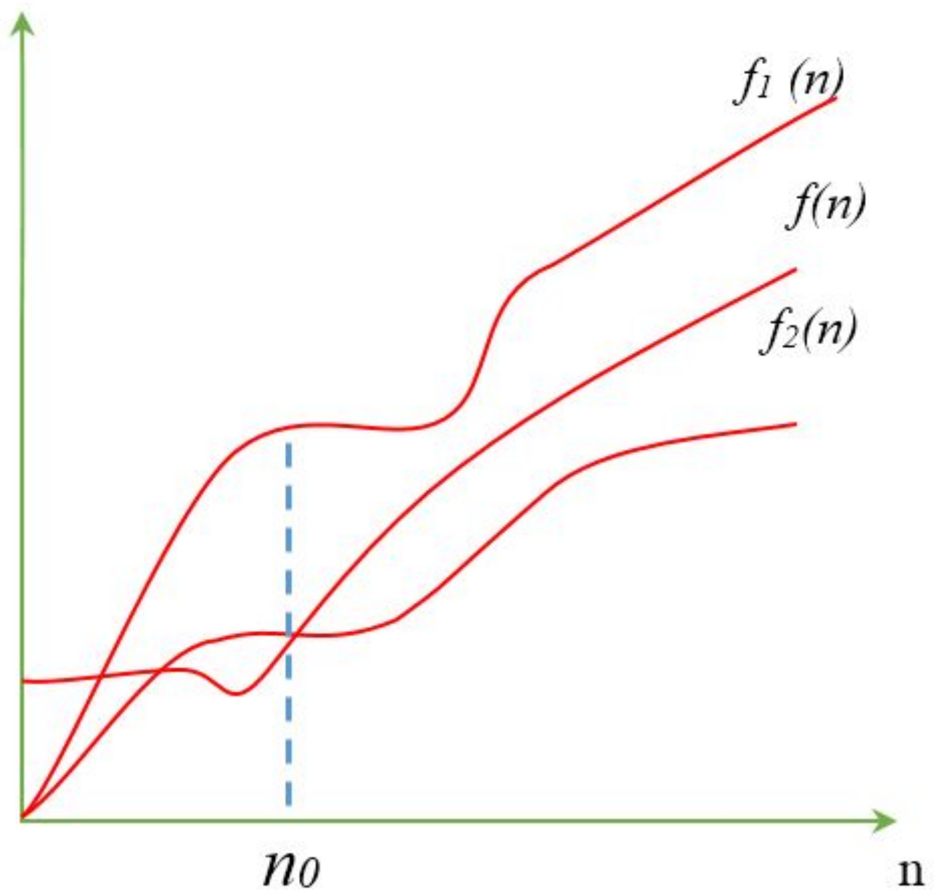
Remember that:

(these can be proven by using calculus)

- Log functions grow more slowly than any power of $\mathtt{n}$ functions, including fractional power.
  i.e. $\log n \in O(n^{\alpha}), \alpha > 0.$
  In fact $\log n \in o(n^{\alpha}), \alpha > 0$
- Power of $\mathtt{n}$ grows more slowly than exponential functions such as 2^n
  i.e. $n^k \in O(2^n)$
  In fact $n^k \in o(2^n)$
- $\log^k n \in O(n)$. In fact, $\log^k n \in o(n)$

Understanding the upper bound & lower bound of a function

- Consider three functions



- For large values of n,
  (i) $f_1(n)$ is larger than f(n), and
  (ii) $f_2(n)$ is smaller than f(n)

  We say that f_1(n) is an *upper bound* for f(n)

  We say that f_2(n) is a *lower bound* for f(n)

- Upper bound is expressed as $O(g(n))$ (“Big-Oh”) and lower bound is expressed as $\Omega(g(n))$ (“Big-Omega”)