Unit 3: Run-Time Analysis Fundamentals

Agenda:

- The Problem:
  - Not all correct codes are the same (Fibonacci)
  - Not all instances are the same (Insertion Sort) (CLRS 24-28)
- Asymptotic Growth of Functions (CLRS Ch.3)
  - Big-O, Big-Ω, Θ, little-o, little-ω
  - Insertion Sort analysis - revised
- Runtime analysis using asymptotic notations

Are all correct codes the same?

- So far — we were given a problem, and we wrote a pseudocode for an algorithm solving it, and we proved the algorithm's correctness.
- Is that enough?
- No. We wish to also argue about the amount of resources the code requires.
  - Time — number of primitive (basic) instructions executed
  - Space — number of memory locations used
  - Energy — A complicated question
  - Random bits
  - ...
- In this course we will often look at Time, seldom Space.
- Is that even important?

Motivation: Fibonacci

- Consider the sequence of Fibonacci numbers defined recursively:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9	10	...
$F(n)$	0	1	1	2	3	5	8	13	21	34	55	...

- Problem: Given n output $F(n)$.

Motivation: Fibonacci

- Direct and easy recursive implementation:

```
procedure fib1(n)
if (n < 2) then
   return n
else
   return fib1(n − 1) + fib1(n − 2)
```

- Non-recursive implementation:

```
procedure fib2(n)
F[1] ← 0
F[2] ← 1
for (j from 3 to n + 1) do
   F[j] ← F[j − 1] + F[j − 2]
return F[n + 1]
```

- Yet another non-recursive implementation:

```
procedure fib3(n)
if (n = 0)
   return 0
x ← 0
y ← 1
for (j from 2 to n)
   newy ← x + y
   x ← y
   y ← newy
return y
```

- Are these calculations all equal?

Motivation: Fibonacci

- Some back-of-the-envelop calculations:
- Let $T_1(n)$ denote the number of recursive calls in `fib1`(n).
  - `fib1`$(n-1)$ — invoked 1 time
  - `fib1`$(n-2)$ — invoked 2 times
  - `fib1`$(n-3)$ — invoked 3 times
  - `fib1`$(n-4)$ — invoked 5 times
  - `fib1`$(n-5)$ — invoked 8 times
  - Claim: `fib1`$(n-i)$ is invoked $F(i+1)$ times for any $1 \leq i \leq n-1$. Proof: induction!
  - It follows $T_1(n) \geq \sum_{i=2}^{n} F(i) \geq F(n)$ which is exponential in n
- Let $T_2(n)$ and $T_3(n)$ denote the number of times we invoke the loop in `fib2` and `fib3` respectively
  - $T_2(n) = T_3(n) = n - 1$
- Let $S_2(n)$ denote the number of "integers stored in memory" in `fib2`.
  - We store an array of all Fibonacci numbers so $S_2(n) \geq n$.
- Let $S_3(n)$ denote the number of "integers stored in memory" in `fib3`.
  - We store x,y, $newy$, maybe a loop counter, maybe program execution counter etc.,but all in all $S_3(n)$ is reasonable constant (say 3).
- In summary:
  - $T_1(n)$ - exponential, $T_2(n), T_3(n)$ - linear
  - $S_2(n)$ - linear, $S_3(n)$ - small constant
- Conclusion: We ♡ `fib3`

Methodologies for analyzing algorithms

- Fine. I'm convinced. We should argue about a code's execution time.
- ... But — How do we measure an algorithm's execution time?
- Several factors involved: implementation language, compiler, operating system, the way it is implemented, test data, computer hardware (CPU, memory, disk, etc), and so on.
- Observation: The running time often increases as the input size increases
- Clever idea: Measure execution time as a function of input size
- Option: Experimental approach — run experiments on different input sizes
- Problems with experimental analysis:
  - We cannot run against all possible inputs
  - Even inputs of the same size may have different running time
  - Some factors (like CPU, memory, implementation, etc) can vary significantly; so test results are very dependent on them.
  - We do not get any insight as to the "bottleneck" of the code
- So we need an analytic way of measuring the running time independent of environment factors (CPU speed, compiler, implementation, etc).

Insertion sort pseudocode (recall)

```
procedure InsertionSort(A, n)    **sort A[1..n] in place
for (j from 2 to n) do
    key ← A[j]             **insert A[j] into sorted sublist A[1..j − 1]
    i ← j − 1
    while (i > 0 and A[i] > key)
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```

- Our goal: Analyze `InsertionSort`'s runtime.
- First approach: each basic command takes some cost (in runtime) per a single execution
- ...and if this command is within a loop — we will multiply it by the number of times the loop runs.
- Let's begin:

proceudre InsertionSort(A)	*cost*	*times*
for (j from 2 to n) do	c_1	n
$\quad key \leftarrow A[j]$	c_2	$n-1$
$\quad i \leftarrow j-1$	c_3	$n-1$
$\quad$ while($i > 0$ and $A[i] > key$)	c_4	???
$\quad \dots$		

Insertion Sort Analysis

<u>`proceudre InsertionSort(A)`</u>	*cost*	*times*
for (j from 2 to n) do	c_1	n
$\quad key \leftarrow A[j]$	c_2	$n-1$
$\quad i \leftarrow j-1$	c_3	$n-1$
$\quad$ while($i > 0$ and $A[i] > key$)	c_4	$\sum_{j=2}^{n} t_j$
$\qquad A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n} (t_j - 1) = \left(\sum_{j=2}^{n} t_j\right) - (n-1)$
$\qquad i \leftarrow i-1$	c_6	$\sum_{j=2}^{n} (t_j - 1) = \left(\sum_{j=2}^{n} t_j\right) - (n-1)$
$\quad A[i+1] \leftarrow key$	c_7	$n-1$

t_j — **<u>instance dependent</u> no. times the `while` loop test is executed for** j**.**
$t_j =$ **number of Key-Comparisons (KC) we make between** $A[j]$ **and elements in** $A[1, ..j-1]$**.**
$t_j = 1+$**number of elements copied to the right (elements bigger than** key**)**

$$T(n) = c_1 n + (c_2 + c_3 - c_5 - c_6 + c_7)(n-1) + (c_4 + c_5 + c_6) \sum_{j=2}^{n} t_j$$

Analysis of insertion sort — Best Case

- So, what should we set as the different values of t_j?
- The optimistic approach: Best Case Analysis (BC)
  - What is a best case instance? (An instance that would make the code run the fastest)
  - Where $t_j = 1$ for any j
  - I.e., we never copy any element to the neighboring right cell
  - I.e., the array is already sorted!

$$\begin{aligned} T_{BC}(n) &= c_1 n + (c_2 + c_3 - c_5 - c_6 + c_7)(n-1) + (c_4 + c_5 + c_6)\sum_{j=2}^{n} t_j \\ &= c_1 n + (c_2 + c_3 - c_5 - c_6 + c_7 + c_4 + c_5 + c_6)(n-1) \\ &= n \cdot (c_1 + c_2 + c_3 + c_4 + c_7) - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

- This means that `InsertionSort`(A, n) will take at least $T_{BC}(n)$ time *on any instance* of size n.

Analysis of insertion sort — Worst Case

- So, what should we set as the different values of t_j?
- The pessimistic approach: Worst Case Analysis (WC)
  - What is the worst case? (An instance that would make the code run the most time)
  - Where we copy all $j-1$ elements to one cell to the right
  - I.e., $t_j = j$ for any j
  - The input is in the reverse order

$$\begin{aligned}
T_{WC}(n) &= c_1 n + (c_2 + c_3 - c_5 - c_6 + c_7)(n-1) + (c_4 + c_5 + c_6)\sum_{j=2}^{n} t_j \\
&= c_1 n + (c_2 + c_3 - c_5 - c_6 + c_7)(n-1) + (c_4 + c_5 + c_6)\sum_{j=2}^{n} j \\
&= c_1 n + (c_2 + c_3 - c_5 - c_6 + c_7)(n-1) + (c_4 + c_5 + c_6)\left(\tfrac{n(n+1)}{2} - 1\right) \\
&= n^2 \cdot \frac{c_4 + c_5 + c_6}{2} \\
&\quad + n \cdot \left(c_1 + c_2 + c_3 + \tfrac{c_4}{2} - \tfrac{c_5}{2} - \tfrac{c_6}{2} + c_7\right) \\
&\quad - (c_2 + c_3 + c_4 + c_7)
\end{aligned}$$

- This means that `InsertionSort`(A, n) will take at most $T_{WC}(n)$ time *on any instance* of size n.

Analysis of insertion sort — Average Case (I)

- So, what should we set as the different values of t_j?

- The intermediate approach: Average-Case Analysis (AC)
  - The input is chosen *randomly*...
  - always ask "average over what input distribution?"
  - One common approach (note: note the only one!) is to assume a uniform distribution on the input.
  - I.e., all inputs, of size n, are equally likely to appear. (Equiprobable)
  - But there are infinitely many possible inputs! (even if A only contains integers)
  - Well... not really.
    We have seen already that the runtime depends only on those t_js...
  - In other words, the algorithm depends solely on the relative comparison between any $A[j]$ and the $j-1$ elements before it.
  - I.e., on the comparison between $A[i]$ and $A[j]$ for any $i \neq j$.
  - That is why we can represent the input as a permutation of n elements:
    The input is $a_{\pi(1)}, a_{\pi(2)}, ..., a_{\pi(n)}$ and the (sorted) output is $a_1, a_2, ..., a_n$ (as we assume $a_1 < a_2 < a_3 < ... < a_n$)
  - So each of the ____ possible inputs is equiprobable
    Each permutation appears with probability ____

Analysis of insertion sort — Average Case (II)

- Average case: always ask "average over what input distribution?"
- We assume uniform distribution
- So what is t_j when the input is chosen uniformly at random (u.a.r) from all possible permutations?
- It is a random variable, depending on the permutation chosen.
- So, on average we take $\mathrm{E}[t_j]$.
- To analyze $\mathrm{E}[t_j]$ we are going to use two properties that are true only for the uniform distribution over permutations.
- First, consider the entire permutation, and ask — what is t_n and $\mathrm{E}[t_n]$?
  - If $A[n]$ is the largest element (a_n) then $t_n = 1$.
  - If $A[n]$ is the second largest elements (a_{n-1}) then $t_n = 2$.
  - If $A[n]$ is the third largest element (a_{n-2}) then $t_n = 3$.
  - ... If $A[n]$ is the smallest element (a_1) then $t_n = n$.
- Claim 1: Let π be a permutation over n elements chosen u.a.r. Then, for any $1 \leq i \leq n$, the probability that the last element of π is the i-th element is $\frac{1}{n}$.
- Proof: Fix i. To get a permutations in which the last element is a_i, we place a_i as the last element, and then place any permutation over the remaining $n-1$ elements in places $\{1, 2, ..., n-1\}$. So the number of permutations where the last element is a_i is $(n-1)!$
  Hence, probability of picking π whose last element is a_i is $\frac{(n-1)!}{n!} = \frac{1}{n}$ $\square$.
- Corollary:
  $\mathrm{E}[t_n] = \sum_{i=1}^{n} (n-i+1) \Pr[\text{last element } = a_i] = \frac{1}{n}(1+2+...+n) = \frac{n+1}{2}$.

Analysis of insertion sort — Average Case (III)

- Average case: always ask "average over what input distribution?"
- We assume uniform distribution and on average we take $\mathrm{E}[t_j]$.
- To analyze $\mathrm{E}[t_j]$ we are going to use two properties that are true only for the uniform distribution over permutations.
- Claim 2: Let π be a permutation over n elements chosen u.a.r. Fix $1 \leq j \leq n$ and a permutation σ over j elements. Then the probability that the permutation over the first j elements of π is precisely σ is $\frac{1}{j!}$.
  In other words, the probability distribution induced on permutations of j elements by taking the first j entries of π is the uniform distribution on j elements.
- Proof: How many permutations are there whose first j elements form exactly σ?
  - Pick the elements that will appear in places $\{1, 2, .., j\}$ ($\binom{n}{j}$ options)
  - The first j elements must appear in the order given by σ
  - The latter $n-j$ elements can appear in any order ($(n-j)!$ options)
  So the probability of picking u.a.r a permutation whose first j entries induce σ is
  $$\frac{1}{n!}\binom{n}{j} \cdot 1 \cdot (n-j)! = \frac{n! \cdot (n-j)!}{n! \cdot j! \cdot (n-j)!} = \frac{1}{j!} \quad \square$$
- Corollary: For any j, $\mathrm{E}[t_j] = \frac{j+1}{2}$
- Proof: We apply the same logic of computing $\mathrm{E}[t_n]$ to the uniform distribution of the permutations on the first j entries.

Analysis of insertion sort — Average Case (IV)

- So, what should we set as the different values of t_j?

- The intermediate approach: Average-Case Analysis (AC)
  - Always ask "average over what input distribution?"
  - We assume uniform distribution
  - Under the uniform distribution, $\mathrm{E}[t_j] = \frac{j+1}{2}$.

$$
\begin{aligned}
T_{uni}(n) &= c_1 n + (c_2 + c_3 - c_5 - c_6 + c_7)(n-1) + (c_4 + c_5 + c_6)\sum_{j=2}^{n} t_j \\
&= c_1 n + (c_2 + c_3 - c_5 - c_6 + c_7)(n-1) + (c_4 + c_5 + c_6)\sum_{j=2}^{n} \tfrac{j+1}{2} \\
&= c_1 n + (c_2 + c_3 - c_5 - c_6 + c_7)(n-1) + \tfrac{c_4+c_5+c_6}{2}\left(\tfrac{(n+1)(n+2)}{2} - 1 - 2\right) \\
&= n^2 \cdot \frac{c_4 + c_5 + c_6}{4} \\
&\quad + n \cdot (c_1 + c_2 + c_3 + \tfrac{3}{4}c_4 - \tfrac{1}{4}c_5 - \tfrac{1}{4}c_6 + c_7) \\
&\quad - (c_2 + c_3 + c_4 + c_7)
\end{aligned}
$$

- This means that `InsertionSort(`A, n`)` will take at least $T_{uni}(n)$ time *on average* if the input is chosen u.a.r from all inputs of size n —
  And this is a **BIG** if...

Analysis of insertion sort — Conclusions

- Best case — gives a lower bound on the runtime of the algorithm
- Worst case — gives an upper bound on the runtime of the algorithm
- Average case — by forcing a distribution over the instances we are making a huge assumption
  - PS. The two claims we had for uniform distribution over permutations are true not just for the last element and for the first j entries.
    - Claim 1 holds for any position i of π, not just the last.
    - Claim 2 holds for any fixed sequence of j coordinates, and not just the first j entries (from 1 to j).
- But all three analyses depend on the 7 unknown constants: $c_1, c_2, ..., c_7$.
- It is laborious to keep specific constants...
- ... but it also doesn't seem right to assume they are all the same...
- ... and it might not be true that each execution of a row takes the same amount of time
- We need some way of saying: $T_{WC}(n)$ and $T_{uni}(n)$ are dominated by n^2 (or: are essentially quadratic); $T_{BC}(n)$ is dominated by n (is essentially linear), without introducing these specific constants (and with them, the assumption of identical runtime per line).
  - For Insertion-Sort: average case roughly as bad as worst case (both are quadratic)
  - It is NOT the case that for all algorithms AC runtime is necessarily similar to the WC runtime
  - For some algorithms — a huge gap between AC runtime and WC runtime.
- Before we just discuss such a way — an important note:

We ♡ worst-case analysis

- Why?
  - A powerful guarantee for all instances!
  - Composes (whereas average-case / best-case do not)
    - If we know the worst-case runtime of alg_1, alg_2 then we can infer (worst-case) runtime of $\langle alg_1, alg_2 \rangle$ (run alg_1 on the input, then run alg_2 on the same input).
    - If we know the worst-case runtime of alg_1, alg_2 then we can infer (worst-case) runtime of $alg_2 \circ alg_1$ (run alg_1 on the input, then run alg_2 on the output of alg_1).
    - If we know the worst-case runtime of alg_1 then we can use it in the analysis of runtime of some alg_2 that uses alg_1 as a subroutine.
  - So given a big-piece of code we can break it to little parts, analyze each part separately, and deduce the overall running time of the entire program.
  - Best-case analysis proves that the runtime of the algorithm on any instance is $\geq$ best-case runtime.
    So BC analysis' roll is to serve as a lower bound for a specific algorithm.
    (And, as a lower bound, it does compose.)

Asymptotic Notation

Asymptotic notation for Growth of Functions: Motivations

- Once upon a time, in a faraway land...
  - ...There was once a problem for which the best algorithm has worst-case running time of $f(n) = n^3$.
    But then a wise scientist managed to find a new algorithm whose running time was $g(n) = 482n^2$.
- Was the wise scientist's effort worth-while?
- The answer: a resounding **YES!**
- It is simple to see that for $n \geq 482$ we have $g(n) \leq f(n)$.
- But what is striking is *how much* faster is the second algorithm in comparison to the former!
n	$f(n)$	$g(n)$	$g(n)/f(n)$
100	1,000,000	4,820,000	4.82
1000	1,000,000,000	482,000,000	0.482
10,000	1,000,000,000,000	48,200,000,000	0.0482
100,000	1,000,000,000,000,000	4,820,000,000,000	0.00482
  - On a computer that does 10^9 operations per second, running alg_2 takes a few thousands of a seconds (i.e., a few days); but running alg_1 takes a million seconds (i.e., about 9 months).

Asymptotic notation for Growth of Functions: Motivations

- To simplify algorithm analysis, want function notation which indicates *rate of growth* (a.k.a., *order* of complexity)
- $O(f(n))$ — **read as "big O of $f(n)$"**
- $\Omega(f(n))$ — **read as "big Omega of $f(n)$"**
- $\Theta(f(n))$ — **read as "Theta of $f(n)$"**
- $o(f(n))$ — **read as "little o of $f(n)$"**
- $\omega(f(n))$ — **read as "little omega of $f(n)$"**

You are expected to recite these definitions in your sleep!

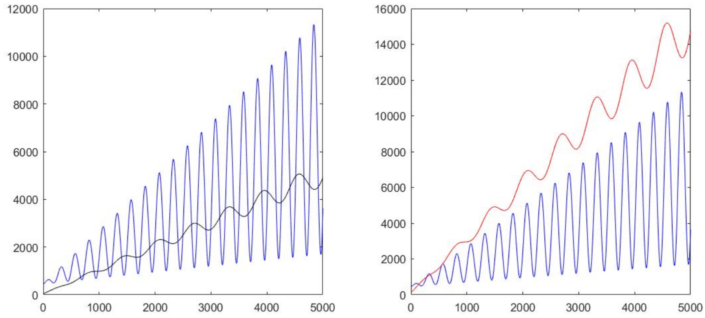
You are expected to understand what these definitions mean!

Big-O Notation: $O(f(n))$

(roughly) The set of functions which, as n gets large, grow no faster than a constant times $f(n)$.

Definition: A function $h(n) : \mathbb{N} \to \mathbb{R}$ belongs to $O(f(n))$ if there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ it holds that $h(n) \leq cf(n)$.

Picture: h and f are drawn in blue and black on the left, resp. After scaling f by 3 (red curve on the right), we can see that for any $n \geq 1000$ we have $h(n) \leq 3f(n)$.



Big-O Notation: $O(f(n))$

(roughly) The set of functions which, as n gets large, grow no faster than a constant times $f(n)$.

Definition: A function $h(n) : \mathbb{N} \to \mathbb{R}$ belongs to $O(f(n))$ if there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ it holds that $h(n) \leq cf(n)$.

examples:

- $482n^2 \in O(n^2)$
  – Set $c = 482$ and $n_0 = 0$ and indeed, $\forall n \geq 0$ we have $482n^2 \leq 482n^2$.
  – Set $c = 1000$ and $n_0 = 10,000$, and $\forall n \geq 10,000$ we have $482n^2 \leq 1000n^2$. – Many other choices of c and n_0.
- $482n^2 \in O(n^3)$
  – Set $c = 482$ and $n_0 = 0$ and indeed, for any $n \geq 0$ we have $482n^2 \leq 482n^3$.
  – Set $c = 1$ and $n_0 = 482$ and we have that for any $n \geq 482$ it holds that $482n^2 \leq 1 \cdot n^3$.
  – Many other choices of c, n_0.
- $15,421n^2 \in O(n^{2.5})$ (Find suitable c, n_0 on your own)
- $(38 + e^5) \cdot n^2 \in O(n^{2.001})$ (Find suitable c, n_0 on your own)
- $n^3 + 255n^2 + n^{2.999} \in O(n^3)$
  – Set $c = 257$ and $n_0 = 0$ and indeed, for any $n \geq 0$ we have
  $$n^3 + 255n^2 + n^{2.999} \leq n^3 + 255n^3 + n^3 = 257 \cdot n^3$$
- $h(n) = \begin{cases} 5^n, & n \leq 10^{120} \\ n^2, & n > 10^{120} \end{cases} \in O(n^2)$
  – Set $c = 1$ and $n_0 = 10^{120} + 1$ and $\forall n \geq n_0$ we have $h(n) \leq 1 \cdot n^2$.

Big-O Notation: $O(f(n))$

(roughly) The set of functions which, as n gets large, grow no faster than a constant times $f(n)$.

Definition: A function $h(n) : \mathbb{N} \to \mathbb{R}$ belongs to $O(f(n))$ if there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ it holds that $h(n) \leq cf(n)$.

examples:

- $n^3 + 482n^2 + 17,200n^{1.5} - 175n + 992n^{0.333} - 253 + \frac{441}{n} \in O(n^3)$
  – Set $c = 20,000$ and $n_0 = 441$ and we have that for any $n \geq 441$

$$n^3 + 482n^2 + 17,200n^{1.5} - 17n + n^{0.333} - 253 + \frac{441}{n}$$
$$\leq n^3 + 482n^3 + 17,200n^3 + 0 + n^3 + 0 + 1$$
$$\leq (1 + 482 + 17,200 + 1 + 1)n^3 \leq 20,000n^3$$

- $1 + 2 + 3 + ... + n \in O(n^2)$.
  – Set $c = 1$ and $n_0 = 1$ and we have that for any $n \geq 1$ it holds

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \leq \frac{n \cdot 2n}{2} = n^2$$

  – Set $c = 1$ and $n_0 = 1$ and for any $n \geq 1$ we have

$$1 + 2 + ... + n \leq n + n + n + ... + n = n \cdot n = n^2$$

Big-O Notation: $O(f(n))$

(roughly) The set of functions which, as n gets large, grow no faster than a constant times $f(n)$.

Definition: A function $h(n) : \mathbb{N} \to \mathbb{R}$ belongs to $O(f(n))$ if there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ it holds that $h(n) \leq cf(n)$.

examples:

- $f(n) = \begin{cases} 5 & , n = 0 \\ f(n-1) + n^2 & , n \geq 1 \end{cases} \in O(n^3)$

  – Set $c = 100$ and $n_0 = 2$ and we prove the required by induction.
  For $n = 2$, $f(2) = f(1) + 2^2 = f(0) + 1^2 + 2^2 = 5 + 1 + 4 \leq 100 \cdot 2^3$
  Fix any n, assuming the required hold for $n-1$ we show it also holds for n. Indeed

  $$\begin{aligned} f(n) &= f(n-1) + n^2 \leq 100 \cdot (n-1)^3 + n^2 \\ &= 100n^3 - 300n^2 + 300n + 100 + n^2 = 100n^3 - 299n^2 + 300n + 100 \end{aligned}$$

  Look at the function $g(x) = -299x^2 + 300x + 100$. Its roots are $\frac{-300 \pm \sqrt{300^2 - 400 \cdot 299}}{-299} \approx \frac{300 \pm 245.15}{299} < 2$. Hence, for any $x \geq 2$ we get $g(x) < 0$. We deduce that

  $$f(n) \leq 100n^3 + g(n) < 100n^3 + 0 = 100 \cdot n^3 \quad \square$$

Big-O Notation: $O(f(n))$

(roughly) The set of functions which, as n gets large, grow no faster than a constant times $f(n)$.

Definition: A function $h(n) : \mathbb{N} \to \mathbb{R}$ belongs to $O(f(n))$ if there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ it holds that $h(n) \leq cf(n)$.

Inverse: A function $h(n) \notin O(f(n))$ if for any $c > 0$ and n_0 there exists $n > n_0$ such that $h(n) > cf(n)$.

Examples:

- $482n^2 \notin O(n)$
  – Fix any $c > 0$ and any $n_0 \in \mathbb{N}$. Since we have $482n^2 > c \cdot n$ iff $n > \frac{c}{482}$, then pick some $n \geq \max\{n_0 + 1, \frac{482}{c}\}$ and for this $n > n_0$ we have $482n^2 > cn$.
- $\frac{1}{482}n^2 \notin O(n^{1.99999})$
  – Fix any $c > 0$ and any $n_0 \in \mathbb{N}$. Since we have $\frac{1}{482}n^2 > c \cdot n^{1.99999}$ iff $n^{0.00001} > 482c$, then pick some $n \geq \max\{n_0 + 1, (482c)^{10000} + 1\}$ and for this $n > n_0$ we have $\frac{1}{482}n^2 > cn^{1.99999}$.

Big-O Notation: $O(f(n))$

(roughly) The set of functions which, as n gets large, grow no faster than a constant times $f(n)$.

Definition: A function $h(n) : \mathbb{N} \to \mathbb{R}$ belongs to $O(f(n))$ if there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ it holds that $h(n) \leq cf(n)$.

Inverse: A function $h(n) \notin O(f(n))$ if for any $c > 0$ and n_0 there exists $n > n_0$ such that $h(n) > cf(n)$.

Examples:

- $n^3 + 255n^2 + n^{2.999} \notin O(n^{2.99999})$
  – Fix any $c > 0$ and any $n_0 \in \mathbb{N}$. We know that $n^3 > cn^{2.99999}$ iff $n^{0.00001} > c$. So set n as $\max\{n_0 + 1, c^{10000} + 1\}$ and we have found some $n > n_0$ for which
  $$n^3 + 255n^2 + n^{2.999} > n^3 > cn^{2.99999}$$
- $n^3 - 255n^2 - n^{2.999} \notin O(n^{2.99999})$
  – Fix any $c > 0$ and any $n_0 \in \mathbb{N}$.
  (1) We have that for $n > 70 \cdot 255$ it holds that $255n^2 < \frac{1}{70}n^3$.
  (2) We also have that for $n > 70^{1000}$ it holds that $n^{2.999} < \frac{1}{70}n^3$.
  (3) We also have that that $\frac{1}{2}n^3 > cn^{2.99999}$ iff $n^{0.00001} > 2c$.
  So set n as any natural $> \max\{n_0, 70 \cdot 255, 70^{1000}, (2c)^{10000}\}$ and for this n, which is $> n_0$, we have
  $$n^3 - 255n^2 - n^{2.999} > n^3 - \frac{1}{70}n^3 - \frac{1}{70}n^3 > \frac{1}{2}n^3 > cn^{2.99999}$$

Big-O Notation: $O(f(n))$

(roughly) The set of functions which, as n gets large, grow no faster than a constant times $f(n)$.

Definition: A function $h(n) : \mathbb{N} \to \mathbb{R}$ belongs to $O(f(n))$ if there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ it holds that $h(n) \leq cf(n)$.

Inverse: A function $h(n) \notin O(f(n))$ if for any $c > 0$ and n_0 there exists $n > n_0$ such that $h(n) > cf(n)$.

This means that for any $c > 0$ there are infinitely many naturals ns $(n_1, n_2, ...)$ such that all of them satisfy $\overline{h(n_i) > cf(n_i)}$.
Q: Why can't there be only finitely many?!?

Examples: $h(n) = \begin{cases} n^2, & n \text{ is even} \\ n^3, & n \text{ is odd} \end{cases} \notin O(n^2)$

– Fix any $c > 0$. Look at all *odd* ns that are greater than c. For any such n (out of these infinitely many ns) we have $h(n) = n^3 > cn^2$.

Definitions:

- $O(f(n))$ is the set of functions $h(n)$ that
  - roughly, grow no faster than $f(n)$, namely
  - Formally: $h(n) \in O(f(n))$ if $\exists c > 0, n_0 \in \mathbb{N}$, such that for all $n \geq n_0$ we have $h(n) \leq cf(n)$.
- $\Omega(f(n))$ is the set of functions $h(n)$ that
  - roughly, grow at least as fast as $f(n)$, namely
  - Formally: $h(n) \in \Omega(f(n))$ if $\exists c > 0, n_0 \in \mathbb{N}$, such that for all $n \geq n_0$ we have $h(n) \geq cf(n)$.
  - $h(n) \in \Omega(f(n))$ if and only if $f(n) \in O(h(n))$
- $\Theta(f(n))$ is the set of functions $h(n)$ that
  - roughly, grow at the same rate as $f(n)$, namely
  - Formally: $h(n) \in \Theta(f(n))$ if $\exists c_0 > 0, c_1 > 0, n_0 \in N$, such that for all $n \geq n_0$ we have $c_0 f(n) \leq h(n) \leq c_1 f(n)$.
  - $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

Definitions (Cont'd):

- $o(f(n))$ is the set of functions $h(n)$ that
  - roughly, grow strictly <u>slower</u> than $f(n)$, namely
  - Formally: $h(n) \in o(f(n))$ if $\lim_{n\to\infty} \frac{h(n)}{f(n)} = 0$
    - I.e. for every $\epsilon > 0$, there exists $n_\epsilon \in \mathbb{N}$ such that for every $n \geq n_\epsilon$ it holds that $\frac{h(n)}{f(n)} < \epsilon$.
    - Including really small values of ϵ (e.g., $10^{-2}, 10^{-9}$ or 10^{-80})
  - Subset of $O(f(n))$, when $f(n) > 0$ for all large enough n

- $\omega(f(n))$ is the set of functions $h(n)$ that
  - roughly, grow strictly <u>faster</u> than $f(n)$, namely
  - Formally: $h(n) \in \omega(f(n))$ if $\lim_{n\to\infty} \frac{h(n)}{f(n)} = \infty$
    - I.e. for every $M > 0$, there exists $n_M \in \mathbb{N}$ such that for all $n \geq n_M$ it holds that $\frac{h(n)}{f(n)} > M$.
    - Including really large values of M (e.g., $10^2, 10^9$ or 10^{80})
  - Subset of $\Omega(f(n))$, when $f(n) > 0$ for all large enough n
  - $h(n) \in \omega(f(n))$ if and only if $f(n) \in o(h(n))$

Note:

- the textbook overloads "="
  - Textbook uses $g(n) = O(f(n))$
  - But we define $O(f(n))$ as a *set* of functions.
  - Both are by now correct
  - My advice: use $g(n) \in O(f(n))$.

Examples:

- Which of the following belongs to $O(n^3)$, $\Omega(n^3)$, $\Theta(n^3)$, $o(n^3)$, $\omega(n^3)$?
  1. $f_1(n) = 19n$
  2. $f_2(n) = 77n^2$
  3. $f_3(n) = 6n^3 + n^2 \log n$
  4. $f_4(n) = 11n^4$

Answers:

1. $f_1(n) = 19n$
2. $f_2(n) = 77n^2$
3. $f_3(n) = 6n^3 + n^2 \log n$
4. $f_4(n) = 11n^4$

- $f_1, f_2, f_3 \in O(n^3)$
  $f_1(n) \leq 19n^3$, for all $n \geq 0$ — $c_0 = 19$, $n_0 = 0$
  $f_2(n) \leq 77n^3$, for all $n \geq 0$ — $c_0 = 77$, $n_0 = 0$
  $f_3(n) \leq 6n^3 + n^2 \cdot n$, for all $n \geq 1$, since $\log n \leq n$
  if $f_4(n) \leq c \cdot n^3$, then for all $n \geq n_0$ we would have $n \leq \frac{c}{11}$ — contra'n
- $f_3, f_4 \in \Omega(n^3)$
  if $f_2(n) \geq c \cdot n^3$, then for all $n \geq n_0$ we would have $\frac{77}{c} \geq n$ — contra'n.
  $f_3(n) \geq 6n^3$, for all $n \geq 1$, since $n^2 \log n \geq 0$
  $f_4(n) \geq 11n^3$, for all $n \geq 0$
- $f_3 \in \Theta(n^3)$ (why?)
- $f_1, f_2 \in o(n^3)$
  $f_1(n)$: $\lim_{n\to\infty} \frac{19n}{n^3} = \lim_{n\to\infty} \frac{19}{n^2} = 0$
  $f_2(n)$: $\lim_{n\to\infty} \frac{77n^2}{n^3} = \lim_{n\to\infty} \frac{77}{n} = 0$
  $f_3(n)$: $\lim_{n\to\infty} \frac{6n^3+n^2 \log n}{n^3} = \lim_{n\to\infty} 6 + \frac{\log n}{n} = 6$
  $f_4(n)$: $\lim_{n\to\infty} \frac{11n^4}{n^3} = \lim_{n\to\infty} 11n = \infty$
- $f_4 \in \omega(n^3)$

Properties of Asymptotic Notation: Reflexivity

- Claim: for any function $f : \mathbb{N} \to \mathbb{R}$ it holds that $f(n) \in O(f(n))$.
  - Same goes for $\Omega(\cdot), \Theta(\cdot)$
- Proof: Given f, set $c = 1, n_0 = 0$ and indeed, for any $n \geq 0$ we have $f(n) \leq 1 \cdot f(n)$. $\square$

Properties of Asymptotic Notation: Additivity

- Claim: for any three functions $f, g, h : \mathbb{N} \to \mathbb{R}$, if $f(n), g(n) \in O(h(n))$ then $f(n) + g(n) \in O(h(n))$.
  - Same goes for *ALL* other notations
- Proof: Given f, g, h, we know that

$$\exists c_1 > 0, n_1 \in \mathbb{N}, \text{ such that for any } n \geq n_1, \text{ we have } f(n) \leq c_1 \cdot h(n)$$
$$\exists c_2 > 0, n_2 \in \mathbb{N}, \text{ such that for any } n \geq n_2, \text{ we have } g(n) \leq c_2 \cdot h(n)$$

- Therefore, for any $n \geq \max\{n_1, n_2\}$ *both* upper-bounds apply!
- Which means that for any $n \geq \{n_1, n_2\}$ we have that

$$f(n) + g(n) \leq c_1 h(n) + c_2 h(n) = (c_1 + c_2) h(n)$$

- Set $c = c_1 + c_2 > 0$ and $n_0 = \max\{n_1, n_2\}$ and we have just shown that

$$\forall n \geq n_0, \qquad f(n) + g(n) \leq c \cdot h(n) \quad \square$$

- Corollary: For any *constant* number of functions $f_1, f_2, ..., f_k : \mathbb{N} \to \mathbb{R}$, if for each i we have $f_i(n) \in O(g(n))$ then $f_1(n) + ... + f_k(n) \in O(g(n))$
- **WARNING:** When the number of summands is NOT a constant and varies with n, the corollary doesn't apply!
- A counter example: let's define n functions $f_1(n) = f_2(n) = ... = f_n(n) = n$. For each i we have $f_i(n) \in O(n)$ (reflexivity), but $f_1(n) + f_2(n) + ... + f_n(n) = n + n + n + ... + n = n^2 \notin O(n)$

Properties of Asymptotic Notation: Multiplicativity

- Claim: for any four functions $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}$, if $f_1(n) \in O(f_2(n))$ and $g_1(n) \in O(g_2(n))$ and all functions take *only positive values*, then $f_1(n) \cdot g_1(n) \in O(f_2 \cdot g_2)$
  - Same goes for *ALL* other notations
- Proof: Given f_1, f_2, g_1, g_2, we know that

$$\exists c_1 > 0, n_1 \in \mathbb{N}, \text{ such that for any } n \geq n_1, \text{ we have } f_1(n) \leq c_1 \cdot f_2(n)$$
$$\exists c_2 > 0, n_2 \in \mathbb{N}, \text{ such that for any } n \geq n_2, \text{ we have } g_1(n) \leq c_2 \cdot g_2(n)$$

- Therefore, for any $n \geq \max\{n_1, n_2\}$ *both* upper-bounds apply!
- Which means that for any $n \geq \max\{n_1, n_2\}$ we have that

$$f_1(n)g_1(n) \leq \big(c_1 f_2(n)\big) \cdot g_1(n) \leq \big(c_1 f_1(n)\big)\big(c_2 g_2(n)\big) = (c_1 \cdot c_2) \cdot f_2(n) g_2(n)$$

Where all inequalities hold because all values of all functions are non-negative.

- Set $c = c_1 \cdot c_2 > 0$ and $n_0 = \max\{n_1, n_2\}$ and we have just shown that

$$\forall n \geq n_0, \quad f_1(n)g_1(n) \leq c \cdot f_2(n)g_2(n) \quad \square$$

Properties of Asymptotic Notation: Transitivity

- Claim: for any three functions $f, g, h : \mathbb{N} \to \mathbb{R}$ that only take non-negative values, if $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ then $f(n) \in O(h(n))$.
  - Same goes for *ALL* other notations
- Proof: Given f, g, h, we know that

$$\exists c_1 > 0, n_1 \in \mathbb{N}, \text{ such that for any } n \geq n_1, \text{ we have } f(n) \leq c_1 \cdot g(n)$$
$$\exists c_2 > 0, n_2 \in \mathbb{N}, \text{ such that for any } n \geq n_2, \text{ we have } g(n) \leq c_2 \cdot h(n)$$

- Therefore, for any $n \geq \max\{n_1, n_2\}$ *both* upper-bounds apply!
- Which means that for any $n \geq \{n_1, n_2\}$ we have that

$$f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n)$$

- Set $c = c_1 \cdot c_2 > 0$ and $n_0 = \max\{n_1, n_2\}$ and we have just shown that

$$\forall n \geq n_0, \quad f(n) \leq c \cdot h(n) \quad \square$$

- BUT if $f(n) \in O(h(n))$ and $g(n) \in O(h(n))$ then f and g aren't comparable...

Properties: Relationships between Notations

- Claim: for any functions $f, g : \mathbb{N} \to \mathbb{R}$, we have that $f(n) \in \Theta(g(n))$ if and only if both (1) $f(n) \in O(g(n))$ and (2) $f(n) \in \Omega(g(n))$ hold.
- Proof: Given f, g, assume that (1) $f(n) \in O(g(n))$ and (2) $f(n) \in \Omega(g(n))$ hold. Which means

$$\exists c_1 > 0, n_1 \in \mathbb{N}, \text{ such that for any } n \geq n_1, \text{ we have } f(n) \leq c_1 \cdot g(n)$$
$$\exists c_2 > 0, n_2 \in \mathbb{N}, \text{ such that for any } n \geq n_2, \text{ we have } f(n) \geq c_2 \cdot g(n)$$

- Therefore, for any $n \geq \max\{n_1, n_2\}$ *both* the upper- and the lower-bound apply!
- Use the same $c_1, c_2 > 0$ and set $n_0 = \max\{n_1, n_2\}$ and we have just shown that

$$\forall n \geq n_0, \quad c_2 g(n) \leq f(n) \leq c_1 g(n) \qquad \Rightarrow \qquad f(n) \in \Theta(f(n))$$

- The opposite direction (staring with assuming the $f(n) \in \Theta(g(n))$ and deriving both big-O and big-Ω) is even more simple and is left as HW. $\square$

Properties: Relationships between Notations

- Claim: for any functions $f, g : \mathbb{N} \to \mathbb{R}$, we have that $f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$.
- Proof: Given f, g, assume $f(n) \in O(g(n))$ then we know

$$\exists c > 0, n_0 \in \mathbb{N}, \text{ such that for any } n \geq n_0, \text{ we have } f(n) \leq c \cdot g(n)$$

- Therefore, for any $n \geq n_0$ we have that $g(n) \geq \frac{1}{c} f(n)$
- Set $c' = 1/c > 0$ and use the same n_0 and we have just shown that

$$\forall n \geq n_0, \quad g(n) \geq c' \cdot f(n) \qquad \Rightarrow \qquad g(n) \in \Omega(f(n))$$

- The opposite direction (staring with assuming the $g(n) \in \Omega(f(n))$ and deriving the big-O) is completely symmetric. $\square$
- Corollary: for any two functions $f, g : \mathbb{N} \to \mathbb{R}$ we have that $f(n) \in \Theta(g(n))$ if and only if we have that (1) $f(n) \in O(g(n))$ and (2) $g(n) \in O(f(n))$.

Properties: Relationships between Notations

- Claim: for any functions $f, g : \mathbb{N} \to \mathbb{R}$ taking positive values, if $f(n) \in o(g(n))$ then $f(n) \in O(g(n))$.
- Proof: Given f, g, assume $f(n) \in o(g(n))$ then we know $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- Which means: for every $\epsilon > 0$ can find some $n_\epsilon \in \mathbb{N}$ such that for any $n \geq n_\epsilon$ it holds that
$$-\epsilon < \frac{f(n)}{g(n)} < \epsilon \qquad (*)$$
- Since $(*)$ holds for any $\epsilon > 0$, it definitely holds for $\epsilon = 1$.
- Namely, there exists some $n_1 \in \mathbb{N}$ such that $\forall n \geq n_1$ we have $\frac{f(n)}{g(n)} < 1$.
- Set $c = 1$ and $n_0 = n_1$ and since $g(n) > 0$ for any n, we have just shown that
$$\forall n \geq n_0, \quad f(n) \leq 1 \cdot g(n) \qquad \Rightarrow \qquad f(n) \in O(g(n)) \quad \square$$

- Claim: for any functions $f, g : \mathbb{N} \to \mathbb{R}$, if we have that $f(n) \in \omega(g(n))$ then it holds that $f(n) \in \Omega(g(n))$.
- Symmetric proof to the little-o case.

Properties: Relationships between Notations

- Claim: for any functions $f, g : \mathbb{N} \to \mathbb{R}$ that take positive values, we have that $f(n) \in o(g(n))$ if and only if $g(n) \in \omega(f(n))$.
- Proof: Given f, g, assume $f(n) \in o(g(n))$, namely that $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- Which means: for every $\epsilon > 0$ can find some $n_\epsilon \in \mathbb{N}$ such that for any $n \geq n_\epsilon$ it holds that
$$-\epsilon < \frac{f(n)}{g(n)} < \epsilon \qquad (*)$$
- What we need to show is that $\lim\limits_{n\to\infty} \frac{g(n)}{f(n)} = \infty$.
  Namely, we need to show that for any $M > 0$ there exists some $n_M \in \mathbb{N}$ such that for any $n \geq n_M$ it holds that $\frac{g(n)}{f(n)} > M$.
- Pick any $M > 0$ arbitrarily. Since $(*)$ holds for any $\epsilon > 0$, it holds for the particular value $\epsilon = \frac{1}{M}$.
- So there exists $n_{1/M}$ such that $\forall n \geq n_{1/M}$ it holds that $0 < \frac{f(n)}{g(n)} < \frac{1}{M}$.
  (The LHS is 0 and not $-\epsilon$ because the two functions are positive.)
- This means that for any $n \geq n_{1/M}$ we have that $\frac{g(n)}{f(n)} > M$.
- Since this holds for an arbitrary $M > 0$, we have that for any $M > 0$ we have a natural (namely $n_{1/M}$) such that $\forall n \geq n_{1/M}, \quad \frac{g(n)}{f(n)} > M$.
- Hence, $\lim\limits_{n\to\infty} \frac{g(n)}{f(n)} = \infty$, i.e. $g(n) \in \omega(f(n))$.
- The proof in the opposite direction (from $\omega(\cdot)$ to $o(\cdot)$) is symmetric. $\square$

Not All Pairs of Functions are Comparable!

- Asymptotic notation isn't a total ordering!
- It is **NOT** true that for any two functions $f, g : \mathbb{N} \to \mathbb{R}$ we either have $f(n) = O(g(n))$ or we have $g(n) = O(f(n))$.
- Example: consider

$$f(n) = \begin{cases} 1, & \text{for odd } n\text{s} \\ n, & \text{for even } n\text{s} \end{cases} \text{ and } g(n) = \begin{cases} n, & \text{for odd } n\text{s} \\ 1, & \text{for even } n\text{s} \end{cases}$$

  then $f(n) \notin O(g(n))$ and $g(n) \notin O(f(n))$.
- Proof: for every $c > 0$ there are infinitely many ns for which $f(n) > c \cdot g(n)$: all <u>even</u> ns satisfying $n > c$.
  For every $c > 0$ there are infinitely many ns for which $g(n) > c \cdot f(n)$: all <u>odd</u> ns satisfying $n > c$.

- And there are other, more complicated examples, of f and g which are monotone, and still aren't comparable in (any) asymptotic notation.

The Limit Rule:

- Claim: for any functions $f, g : \mathbb{N} \to \mathbb{R}$ that take positive values, if we have that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ **exists**, then

$$\text{if } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} \infty & \text{, then } f(n) \in \omega(g(n)) \subset \Omega(g(n)) \\ L & \text{for some real } L > 0, \text{ then } f(n) \in \Theta(g(n)) \\ 0 & \text{, then } f(n) \in o(g(n)) \subset O(g(n)) \end{cases}$$

- Proof: The first and third case follow from the definitions. We only prove the second case.
- If the limit$= L$ then for every $\epsilon > 0$ can find some $n_\epsilon \in \mathbb{N}$ such that for any $n \geq n_\epsilon$ it holds that

$$L - \epsilon < \frac{f(n)}{g(n)} < L + \epsilon \qquad (*)$$

- So specifically, for $\epsilon = \frac{L}{2}$ we have that some $n_{L/2}$ exists such that

$$\text{for any } n \geq n_{L/2}, \qquad \frac{L}{2} \cdot g(n) < f(n) < \frac{3L}{2} \cdot g(n)$$

  because $g(n)$ is positive.
- Set $c_1 = \frac{3L}{2} > 0, c_2 = \frac{L}{2} > 0$ and $n_0 = n_{L/2}$ and we have that $f(n) \in \Theta(g(n))$. $\square$

Handy 'big O' tips: Logarithm

- If $f, g : \mathbb{N} \to \mathbb{R}$ are both positive functions then $f(n) \geq g(n)$ iff $2^{f(n)} \geq 2^{g(n)}$.
  - E.g, because $\forall n, n \leq 2^n$ then $\forall n \geq 1$, $\log(n) \leq n$. So $\log(n) \in O(n)$.
- It is often very useful to write $f(n) = 2^{\log(f(n))}$.
- Claim: for any functions $f, g : \mathbb{N} \to \mathbb{R}$ that take positive values, denote $a_n = \log(f(n)) - \log(g(n))$.
  If $\lim_{n\to\infty} a_n$ exists, then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 2^{\lim_{n\to\infty} a_n}$, and we have:

  $$\text{if } \lim_{n\to\infty} a_n = \begin{cases} \infty & \text{, then } f(n) \in \omega(g(n)) \subset \Omega(g(n)) \\ L & \text{for some real } L \text{, then } f(n) \in \Theta(g(n)) \\ -\infty & \text{, then } f(n) \in o(g(n)) \subset O(g(n)) \end{cases}$$
- Note the difference from the limit rule in the 2nd and 3rd cases.

Handy 'big O' tips: Applying Functions

- Suppose $f, g, h : \mathbb{N} \to \mathbb{R}$ are both positive functions and h is unbounded and monotone increasing.
  –If $\exists n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ we have $f(n) \geq g(n)$ then $\exists n_0'$ such that for all $n \geq n_0'$ we have $f(h(n)) \geq g(h(n))$.
  –If $\exists n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ we have $f(n) \geq g(n)$ then $\exists n_0'$ such that for all $n \geq n_0'$ we have $h(f(n)) \geq h(g(n))$.
- Example: $n \in o(n^2)$ and therefore $\log(n) \in o((\log(n))^2)$.
  - $f(n) = n$, $g(n) = n^2$ and $h(n) = \log(n)$.
- Example: $n^2 \in o(n^3)$ and therefore $2^{2n} \in o(2^{3n})$.
  - $f(n) = n^2$, $g(n) = n^3$ and $h(n) = 2^n$.
- Example: $n^2 \in O(2^n)$ and therefore $\log(n) \in O(\sqrt{n})$.
  - First, $f(n) = n^2$, $g(n) = 2^n$ and $h(n) = \sqrt{n}$.
    Hence, $h(f(n)) = n$ and $h(g(n)) = 2^{n/2}$ so we have that $n \in O(2^{\frac{n}{2}})$.
  - Now we use $f(n) = n$, $g(n) = 2^{n/2}$ and $h(n) = \log(n)$.
    So $f(h(n)) = \log(n)$ and $g(h(n)) = \sqrt{n}$.

Logarithmic vs. Polynomial

- Well, first we argue that for any n we have $n \leq 2^n$.
  - Prove by induction. Base: $0 \leq 1$. Ind step: for any n we have $n+1 \leq 2^n + 1 \leq 2^n + 2^n = 2 \cdot 2^n = 2^{n+1}$.
- Therefore, apply the monotone function $\log(n)$ we get: for any $n \geq 1$ we have $\log(n) \leq n$. Thus, $\log(n) \in O(n)$.
- Moreover, apply this result to the monotone function $\sqrt{n}$. We get that for any $n \geq 1$ it holds that $\frac{1}{2}\log(n) = \log(\sqrt{n}) \leq \sqrt{n}$. So, $\log(n) \in O(\sqrt{n})$.
- In fact, fix any $\epsilon > 0$. We use the same reasoning to argue that for any $n \geq 1$ it holds that $\epsilon \log(n) = \log(n^\epsilon) \leq n^\epsilon$. Hence, $\log(n) \leq \frac{1}{\epsilon} n^\epsilon$ for any $n \geq 1$, so $\log(n) \in O(n^\epsilon)$ for any fixed constant $\epsilon > 0$.
- Now we argue that for any $\epsilon > 0$ and $k > 0$ we have $(\log(n))^k \in O(n^\epsilon)$. Fix ϵ and k. We know that $\log(n) \in O(n^{\frac{\epsilon}{k}})$. So exists $c > 0$ and n_0 such that for any $n \geq n_0$ it holds: $\log(n) \leq c n^{\frac{\epsilon}{k}}$; which means that $(\log(n))^k \leq c^k \cdot n^\epsilon$. Hence $(\log(n))^k \in O(n^\epsilon)$.
- In fact, we can now argue that for any $\epsilon > 0$ and any $k > 0$ we have that $(\log(n))^k \in o(n^\epsilon)$.
  Fix ϵ and k. By the previous claim, $(\log(n))^k \in O(n^{\frac{\epsilon}{2}})$, so exists some $c > 0$ and n_0 such that for any $n \geq n_0$ we have $(\log(n))^k \leq c \cdot n^{\frac{\epsilon}{2}}$.
  Therefore, for large enough n we have

$$\frac{\log(n)^k}{n^\epsilon} \leq c \cdot \frac{n^{\frac{\epsilon}{2}}}{n^\epsilon} = \frac{c}{n^{\frac{\epsilon}{2}}} \overset{n\to\infty}{\to} 0$$

Polynomial vs. Exponential

- We know: For any $\epsilon > 0$ we have that $\log(n) \in o(n^\epsilon)$.
- In particular, we know that for any $\epsilon > 0$ and any $k > 0$ we have that for large enough ns we get $\log(n) \leq \frac{1}{k} \cdot n^\epsilon$.
- Hence for large enough ns we have $n^k = 2^{k\log(n)} \leq 2^{n^\epsilon}$ for any $\epsilon > 0$ and any $k > 0$. Namely, $n^k \in O(2^{n^\epsilon})$.
- In particular, for a given $\epsilon > 0, k > 0$ we have that for all large enough ns we have $n^k \leq 2^{n^{\frac{\epsilon}{2}}}$. Thus,

$$\frac{n^k}{2^{n^\epsilon}} \leq \frac{2^{n^{\frac{\epsilon}{2}}}}{2^{n^\epsilon}} = 2^{n^{\epsilon/2} - n^\epsilon} = 2^{n^{\epsilon/2} \cdot (1 - n^{\epsilon/2})}$$

  Note that $n^{\epsilon/2} \to \infty$ so for large enough ns we clearly have $n^{\epsilon/2} > 2$. So for large enough ns we have

$$\frac{n^k}{2^{n^\epsilon}} \leq 2^{n^{\epsilon/2} \cdot (1 - n^{\epsilon/2})} \leq 2^{n^{\epsilon/2}(1-2)} = 2^{-n^{\epsilon/2}} = \frac{1}{2^{n^{\epsilon/2}}} \xrightarrow{n \to \infty} 0$$

- Conclusion: The "Logarithmic $\ll$ Polynomial $\ll$ Exponential" rule. For any $\epsilon > 0$ and any $k > 0$ we have that

$$\log(n)^k \in o(n^\epsilon) \text{ and } n^k \in o(2^{n^\epsilon})$$

Logarithmic $\ll$ Polynomial $\ll$ Exponential

- The "Logarithmic $\ll$ Polynomial $\ll$ Exponential" rule.
  For any $\epsilon > 0$ and any $k > 0$ we have that
  $$\log(n)^k \in o(n^\epsilon) \text{ and } n^k \in o(2^{n^\epsilon})$$
- Make sure you understand the statement.
- Note what it doesn't mean!
  It doesn't mean that whenever you see $\log(\cdots)$ then it will always be the small term, and when you see $2^{\cdots}$ it doesn't mean that it is the large term.
- For example, $\log(n) \notin o(2^{\log\log\log(n)})$. (In fact, $\log(n) \in \omega(\log\log(n))$.)
- And we also have $2^{\sqrt{\log(n)}} \in o(n)$. (Prove it!)

The Class $O(1)$

- By definition, $f : \mathbb{N} \to \mathbb{R}$ belongs to $O(1)$ if there exists $c > 0$ and n_0 such that for any $n \geq n_0$ we have $f(n) \leq c$.
- Therefore, for any n, we have $f(n) \leq \max\{f(1), f(2), .., f(n_0), c\}$.
- In other words, if $f \in O(1)$ then there exists $M > 0$ such that for any n we have $f(n) \leq M$. Namely, f is (upper) bounded.
- Clearly, if f is (upper) bounded then $f \in O(1)$ (set c as the bound M).
- In other words, $O(1)$ is the set of bounded functions.

- So, from now on, whenever a computation involves only a constant number of instructions, we will say it runs in $O(1)$-time.
- This means, that the following implementations are both correct and take $O(1)$-time.

```
procedure Swap(a, b)
temp ← a
a ← b
b ← temp
```

```
procedure Swap(a, b)
temp ← nil
temp ← b
temp ← a
a ← temp
a ← b
b ← temp
```

- And they will be equivalent to any correct code for `Swap` even if it has $1,000,000$ instructions...
- Q: What is the set of functions $o(1)$?

logarithm review:

For any $b > 1$ and $n > 0$ we define

- Definition of $\log_b(n)$: $b^{\log_b n} = n$
- $\log_b n$ as a function in n: increasing, one-to-one
- $\ln n = \log_e n$ (natural logarithm)
- $\lg n = \log_2 n$ (base 2, binary)

- $\log_b 1 = 0$
- For any x and any p, $\log_b x^p = p \log_b x$
- For any x and any y, $\log_b(xy) = \log_b x + \log_b y$
- For any x and any y, $x^{\log_b y} = y^{\log_b x} = b^{\log_b(x) \cdot \log_b(y)}$
- For any x and any $c > 1$, $\log_b x = (\log_b c)(\log_c x)$
- For any $b > 1$ we have $\Theta(\log_b n) = \Theta(\log n)$
- $(\log n)^k \in o(n^\epsilon)$, for any fixed positives k and ϵ

Logarithms and the Harmonic Number:

- The derivative: $\frac{d}{dx}\ln x = \frac{1}{x}$
- We denote the harmonic number $H(n) = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$.
- Now clearly, for every $k \geq 1$, and any $x \in (k-1, k]$ and any $y \in [k, k+1]$ we have $x \leq k \leq y$ so $\frac{1}{y} \leq \frac{1}{k} \leq \frac{1}{x}$.
- Therefore we have $\int\limits_{x=k}^{k+1} \frac{1}{x}dx \leq \frac{1}{k} \leq \int\limits_{x=k-1}^{k} \frac{1}{x}dx$
- We thus have for any $n \geq 1$:

$$H(n) = \left(1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n-1}\right) + \frac{1}{n} \geq \left(\int_{x=1}^{n} \frac{1}{x}dx\right) + 0 = \ln(n)$$

$$H(n) = 1 + \left(\frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n-1} + \frac{1}{n}\right) \leq 1 + \left(\int_{x=1}^{n} \frac{1}{x}dx\right) = \ln(n) + 1$$

  and we deduce that $H(n) = \Theta(\log(n))$, or even: $H(n) = \ln(n) + O(1)$.
- In fact, it is known that $|H(n) - \ln(n)| \to_n 0.57...$ (Euler constant)

Tower of Exponents

- Define $f(n) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}_{n \text{ times}}$
- So $f(0) = 1$, $f(1) = 2$, $f(2) = 4$...
- $f(3) = 16$, $f(4) = 65,536$, $f(5)$ has more than $19,500$ digits!
- REALLY fast growing function.

$\log^*$ function

- The inverse of the tower of exponent.
- Formally: $\log^*(n) = \min\{k : \ 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}_{k \text{ times}} \geq n\}$
- Alternatively: $\min\left\{k : \ \underbrace{\lg\lg\lg\ldots\lg}_{k}(n) \leq 0\right\}$
- REALLY slow growing function.
  - In fact, seeing as $\log^*(10^{80}) = 5$, it is safe to say that 5 is an upper bound on all realistic instances.
  - Nonetheless, $\lim_n \log^*(n) = \infty$, and so $\log^*(n) \notin O(1)$
  - whereas the constant function $f(n) = 10000$ does belong to $O(1)$.
  - and clearly, $5 < 10000$...

Another useful formula is Stirling's Approximation: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.
Example: The following functions are ordered in increasing order of growth (each is in big-Oh of next one). Those in the same group are in big-Theta of each other.

$$\{n^{1/\log n},\quad 1\},\ \ \log^*(n),\quad \{\log\log n,\quad \ln\ln n\},\quad \sqrt{\log n},\quad \ln n,\quad \log^2 n,$$

$$2^{\sqrt{\log n}},\quad (\sqrt{2})^{\log n},\quad 2^{\log n},\quad \{n\log n,\quad \log(n!)\},\quad n^2,\quad \{n^3,\quad 8^{\log(n)}\}$$

$$(\log n)!,\quad \{(\log n)^{\log n},\quad n^{\log\log n}\},\quad \left(\frac{3}{2}\right)^n,$$

$$2^n,\quad n\cdot 2^n,\quad e^n,\quad n!,\quad (n!)^2,\quad (n^2)!,\quad 2^{2^n},\quad 2^{2^{\cdot^{\cdot^{\cdot^2}}}}\Big\}_{n\ \text{times}}$$

Back to the design of algorithms...

Runtime Analysis using Big-O Notation

- <u>procedure fib2(n)</u>
  $F[1] \leftarrow 0$
  $F[2] \leftarrow 1$
  for (j from 3 to $n+1$) do
      $F[j] \leftarrow F[j-1] + F[j-2]$
  return $F[n+1]$
- Runtime analysis — big-O:
  - First two instructions take $O(1)$-time.
  - Each loop iteration involves only a constant number of instructions, so it takes $O(1)$-time.
  - We repeat the loop for $O(n)$ iterations.
  - After the loop we do another constant number of instruction, so $O(1)$-time
  - Overall runtime: $O(1) + O(n) \cdot O(1) + O(1) = O(n)$.
- Runtime analysis — big-Ω:
  - Inside the loop, we do at least one instruction - takes at least one clock-tic
  - The loop iterated $n-1$ times
  - So our runtime is at least $n - 1 = \Omega(n)$.
- Conclusion: runtime is $\Theta(n)$.

Runtime Analysis using Big-O Notation

-
```
procedure InsertionSort(A, n)   **sort A[1..n] in place
for (j from 2 to n) do
    key ← A[j]          **insert A[j] into sorted sublist A[1..j − 1]
    i ← j − 1
    while (i > 0 and A[i] > key)
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```
- Runtime analysis, WC — big-O:
  - The for-loop iterates $O(n)$ times
  - In each iteration: we do constant amount of instructions + the while loop.
  - The while-loop iterates at most $O(n)$ times.
  - Overall runtime: $O(n)\,(O(1) + O(n)) = O(n^2)$.
- Runtime analysis, WC — big-Ω:
  - As discussed, in the WC, for every j, the while-loop iterates j times.
  - This means that for each $j \in \{\frac{n}{2}, \frac{n}{2} + 1, ..., n - 1, n\}$ the while loop iterates at least $j \geq \frac{n}{2}$ times, and in each iteration we so something (take at least one action).
  - So our runtime is at least $\frac{n}{2} \cdot \frac{n}{2} = \Omega(n^2)$.
- Conclusion: WC runtime is $\Theta(n^2)$.

Runtime Analysis using Big-O Notation

- procedure InsertionSort(A, n) **sort $A[1..n]$ in place

```
for (j from 2 to n) do
    key ← A[j]            **insert A[j] into sorted sublist A[1..j − 1]
    i ← j − 1
    while (i > 0 and A[i] > key)
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```

- Runtime analysis, BC — big-O:
  - The for-loop iterates $O(n)$ times
  - In each iteration: we do constant amount of instructions + the while loop.
  - In the BC – the while-loop iterates at most $O(1)$ times.
  - Overall runtime: $O(n)\,(O(1) + O(1)) = O(n)$.
- Runtime analysis, BC — big-Ω:
  - The for-loop iterates $\Omega(n)$ times, each time involves a non-empty set of instructions, so it takes $\Omega(1)$ time.
- Conclusion: BC runtime is $\Theta(n)$.

Runtime Analysis using Big-O Notation

- Note how we can assess the best-case and worst-case runtimes in terms of both big-O and big-Ω (and any other asymptotic notation).
- **WARNING:** A common mistake would be to assume that we can only do big-O analysis for the worst-case runtime and only big-Ω analysis for the best-case runtime.
  That is a false assumption!
- Big-O / big-Ω etc. are properties of functions. ANY function.
- The function we describe in those asymptotic notation are often the runtimes of the algorithm, but we could focus on the WC-runtime or the BC-runtime and analyze each one's asymptotic growth.
- What is true is that if we manage to show that WC-runtime(alg)$\in O(f(n))$ and BC-runtime(alg)$\in \Omega(g(n))$ then asymptotically the runtime of alg on *any* instance of size n will be lower-bounded by something proportional to $g(n)$ and upper-bounded by something proportional to $f(n)$.
- And if we lucked out and WC-runtime(alg)$\in O(f(n))$ and BC-runtime(alg)$\in \Omega(f(n))$ asymptotically the runtime of alg on *any* instance of size n will be proportional to $f(n)$.

Runtime Analysis using Big-O Notation

- Lastly, let's get back to our Arrays vs. Linked Lists comparison, and compare the runtime of different operations:

Operation	Array	List
Insert(x)	$\Theta(1)$	$\Theta(1)$
Delete(x)	$\Theta(1)$	$\Theta(1)$ (with a pointer to x)
Access k-th element	$\Theta(1)$	$\Theta(k)$ if doubly linked: $\Theta\left(\min\{k, n-k\}\right)$
Find(x)	WC: $\Theta(n)$, BC: $\Theta(1)$	WC: $\Theta(n)$, BC: $\Theta(1)$
Merge(A, B)	$\Theta(A.size)$ or $\Theta(B.size)$	$\Theta(1)$ (with pointer to $tail$)

- How long does it take to convert an array into a list? How long for converting a list into an array?
- HW – write code that takes an array of size n and creates a list with the same elements and in the same order and runs in time $\Theta(n)$.
  HW – write code that takes a list of n elements of the same type and creates an array with the same elements and in the same order and runs in time $\Theta(n)$.

Summary

- Arguing about the amount of resources an algorithm takes is a must
  - Since for the same problem there could be many algorithm taking different runtime / space /...
- You may decide your analysis is BC, WC or AC (and on what distribution)...
- ... But we tend to prefer WC analysis, as it *composes* and gives a strong upper-bound on all instances.
- Rigorous analysis is tedious, and often not insightful, so we turn to a notation that expresses the asymptotic growth of the runtime as a function of the input size.
- Knowing precisely what the 5 asymptotic notations mean — is imperative for any analysis we will do in class, and that you will do in your professional life.
- And indeed, using this notation, arguing about the runtime of a code (with loops) becomes much simpler.
- Runtime analysis for a code that uses recursions — next week