Lecture 2 : More examples of big-Oh and the Master Theorem

Justin Pearson — Based on Slides by Pierre Flener

Overview

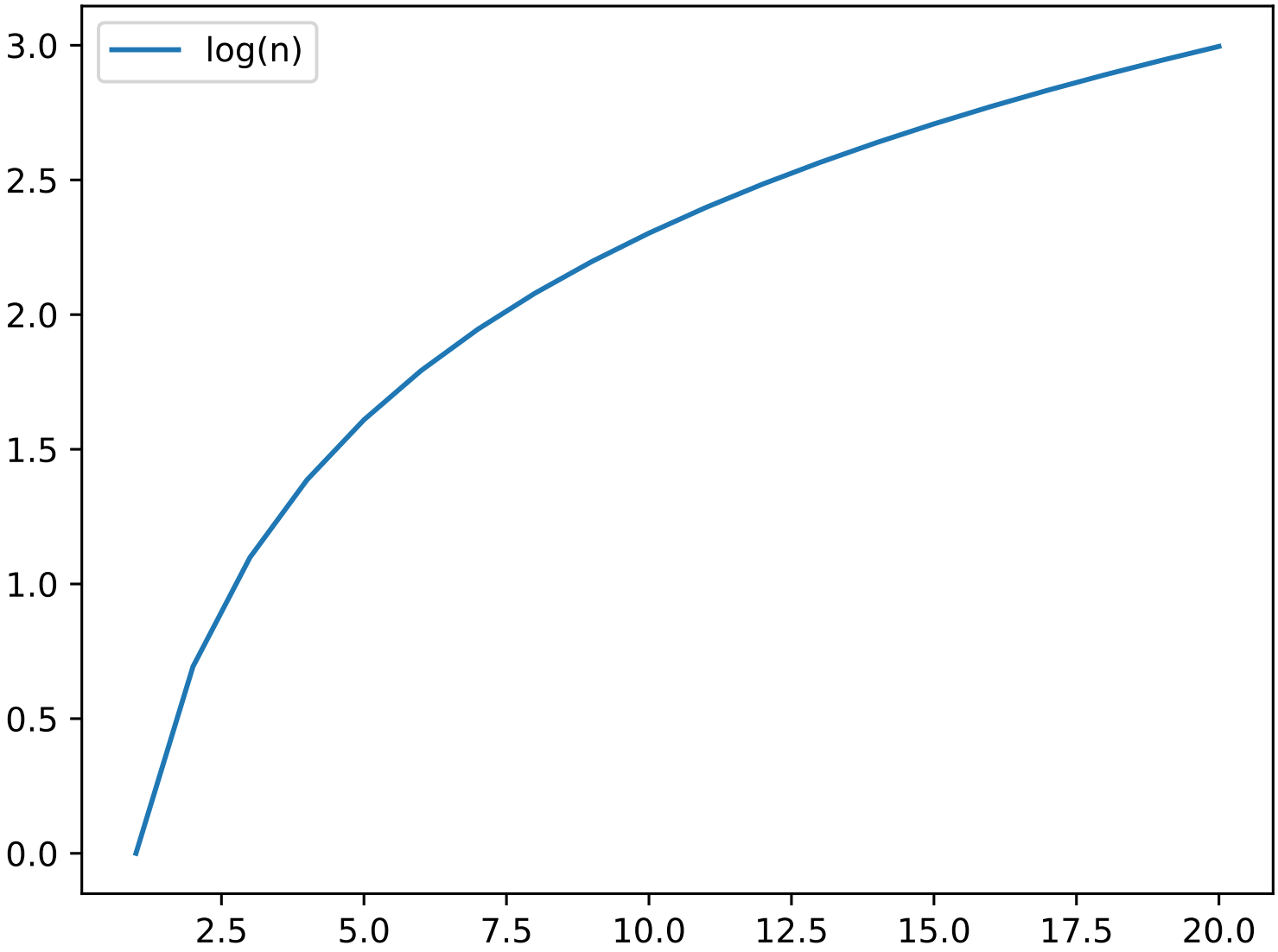
- Some terminology for function growth and complexity classes.
- Some examples of run time analysis
- The Master Theorem, how to use it and a hint of the proof.
- Revision of some of the assumptions of complexity analysis.

Terminology

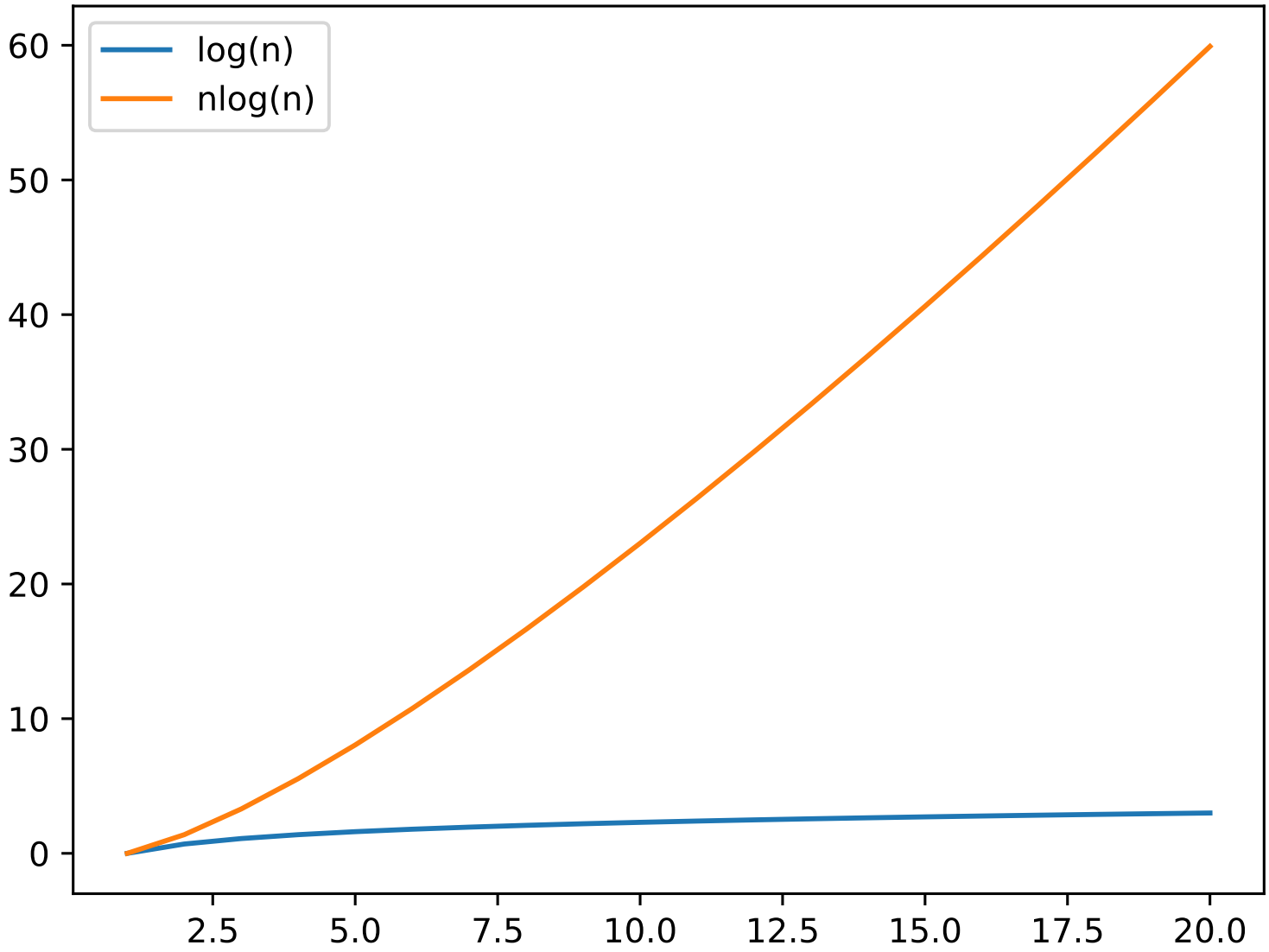
Let n denote the input size then we have the following table in order of complexity.

Function	Growth Rate	
1	constant	sub-linear
$\log n$	logarithmic	
$\log^2 n$	log-squared	
n	linear	polynomial
$n \cdot \log n$		
n^2	quadratic	
n^3	cubic	
k^n $(k \geq 2)$	exponential	exponential
$n!$		super-exponential
n^n		

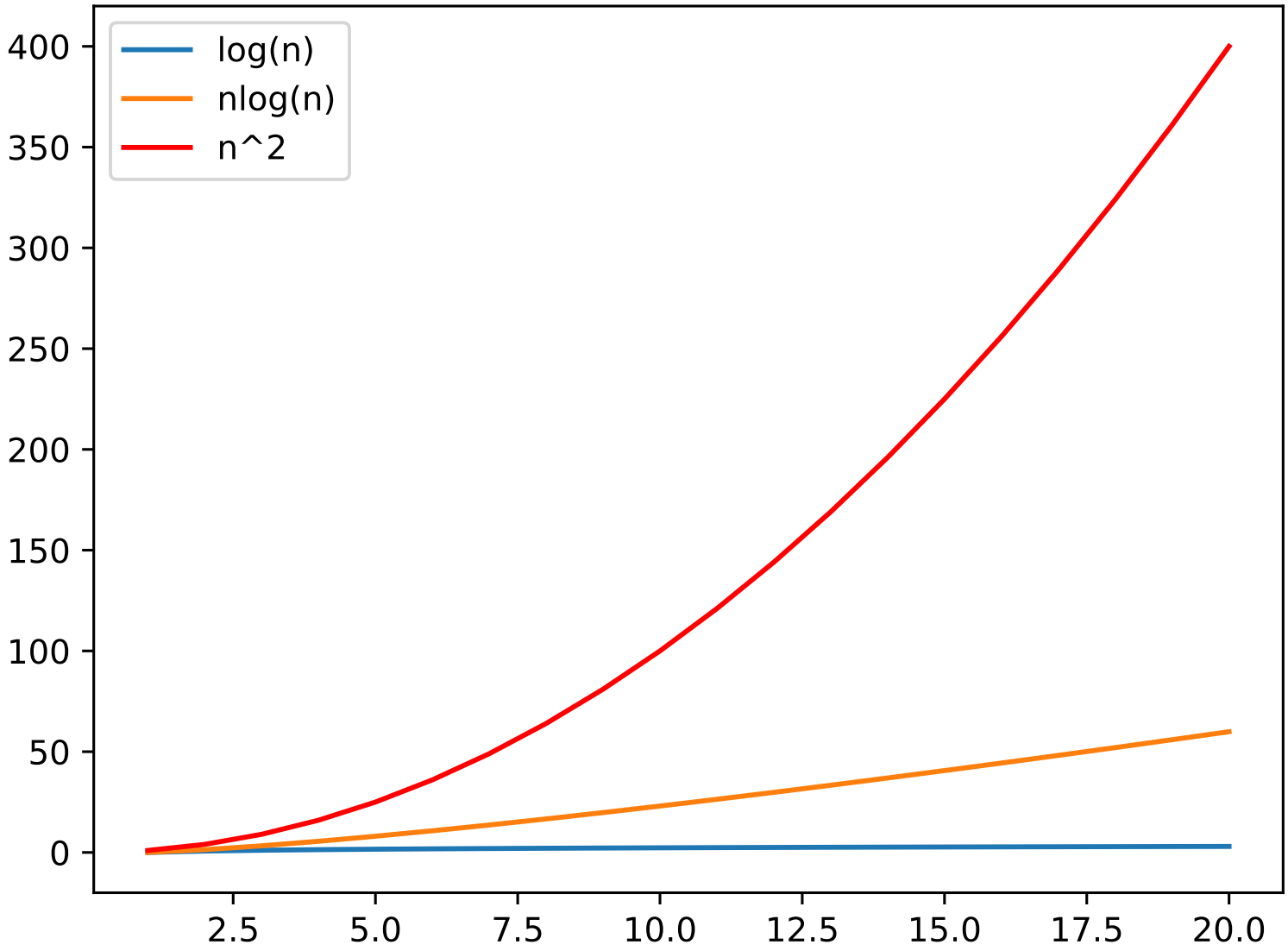
Comparing functions



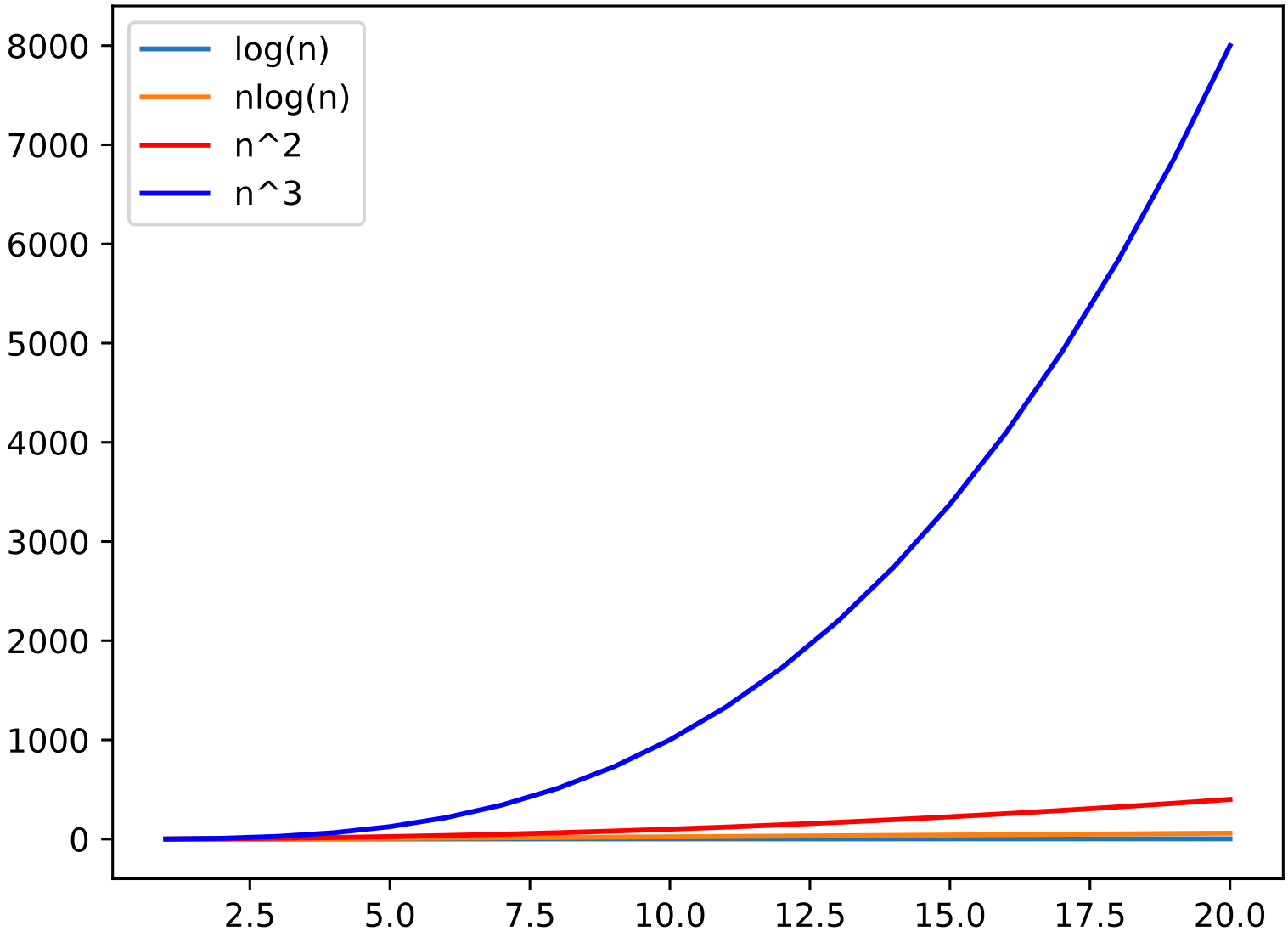
Comparing functions



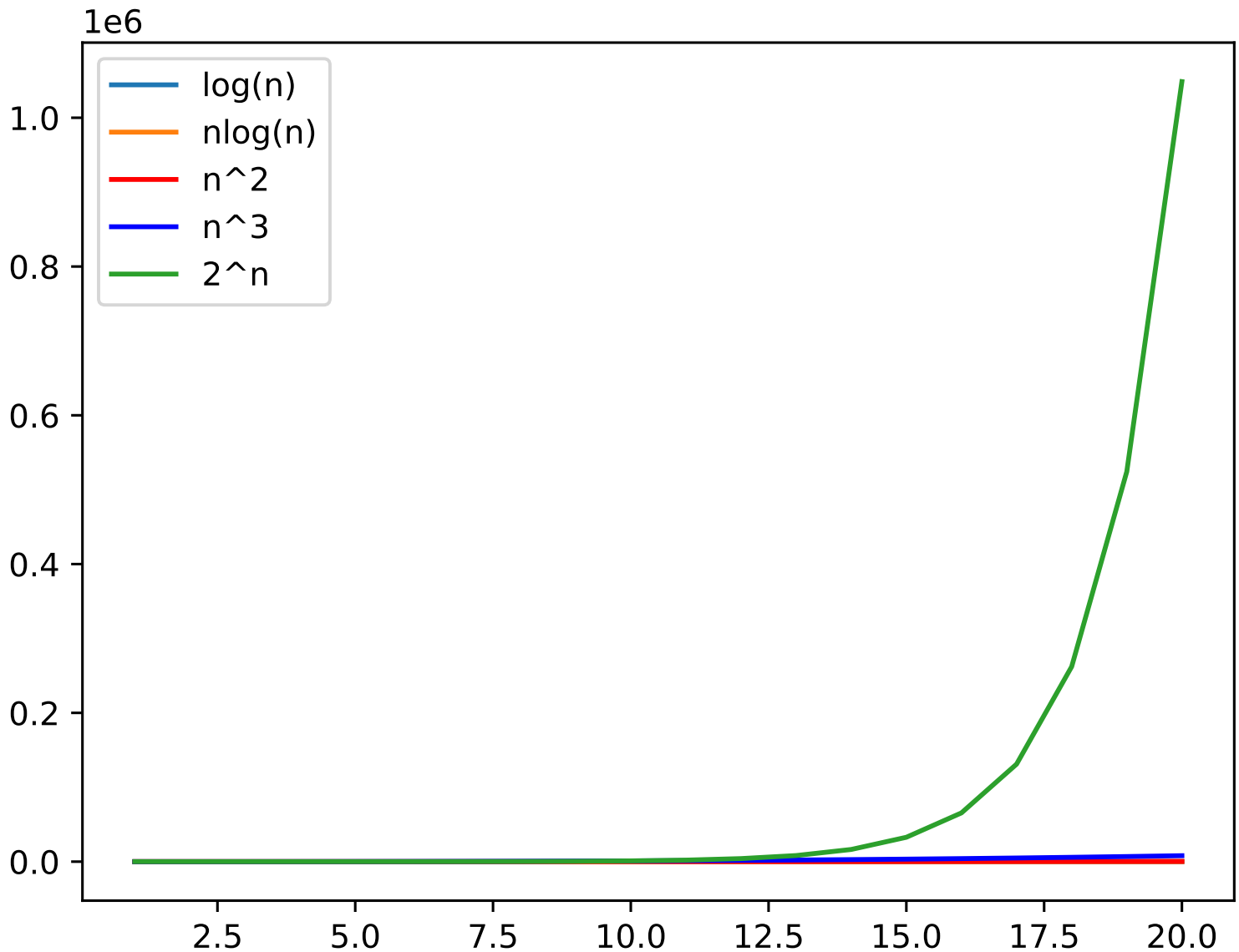
Comparing functions



Comparing functions



Comparing functions



Growth

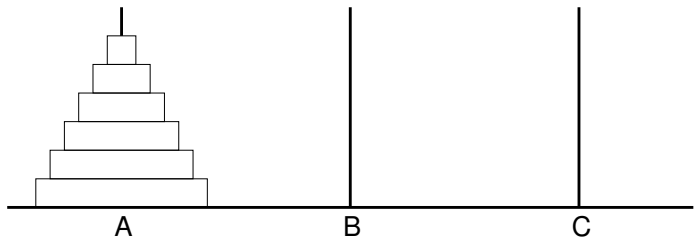
Let n denote the input size then we have the following table in order of complexity.

Function	Growth Rate	
1	constant	sub-linear
$\log n$	logarithmic	
$\log^2 n$	log-squared	
n	linear	polynomial
$n \cdot \log n$		
n^2	quadratic	
n^3	cubic	
k^n $(k \geq 2)$	exponential	exponential
$n!$		super-exponential
n^n		

Even for small values of n, n^3 can grow quite fast, if you algorithm has the worst time complexity of 2^n then you are in trouble.

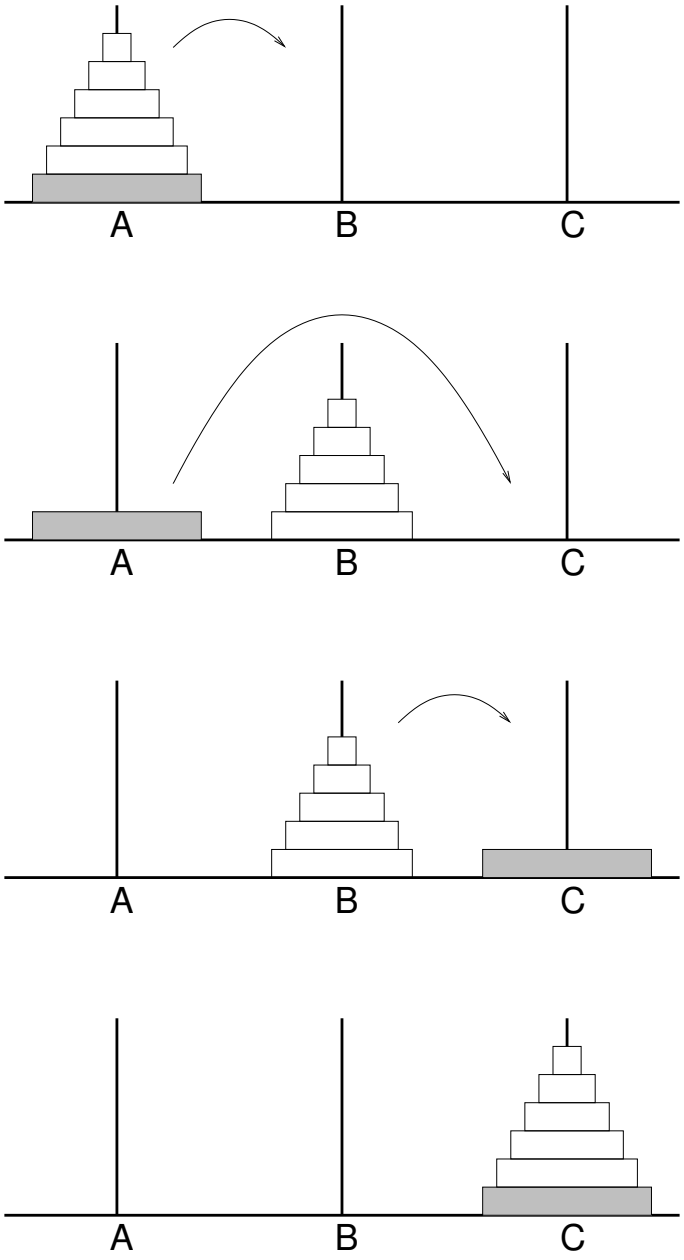
Example: Towers of Hanoi

Many legends, origins in India.
Initial state: Tower A has n disks stacked by decreasing diameter.
Towers B and C are empty.



Rules
– Only move one disk at a time.
– Only move the top-most disk of a tower.
– Only move a disk onto a larger disk (if any).
Objective and final state: Move all the disks from tower A to tower C, using tower B, without violating any rules.
Problem: What is a (minimal) sequence of moves to be made for reaching the final state from the initial state, without violating any rules.

Hanoi: Strategy



1. Recursively move $n-1$ disks from tower A to tower B, using tower C.

2. Move one disk from tower A to tower C.

3. Recursively move $n-1$ disks from tower B to tower C, using tower A.

Hanoi: Specification and Program

```
--- hanoi (n, from, via, to)

fun hanoi (0, from, via, to) = ""
  | hanoi (n, from, via, to) =
    hanoi (n-1, from, to, via) ^ from ^ "->" ^ to ^ " " ^
    hanoi (n-1, via, from, to)
```

Will the end of the world be provoked by the call

```
hanoi (64, "A", "B", "C")
```

even on the fastest computer of 20 years from now?!

Hanoi: Analysis

Let $M(n)$ be the number of moves that must be made for solving the problem of the Towers of Hanoi with n disks.

From the program, we get the recurrence:

$$M(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 \cdot M(n-1) + 1 & \text{if } n > 0 \end{cases} \tag{1}$$

How to solve this recurrence?
Guess the closed form and prove it!

- Guessing: By expansion method, iterative / substitution method, or recursion-tree method.
- Proving: By induction,
  or by application of a pre-established formula.

Hanoi: Iterative / Substitution Method

$$
\begin{aligned}
M(n) &= 2 \cdot M(n-1) + 1, \text{ by the recurrence (1)} \\
&= 2 \cdot (2 \cdot M(n-2) + 1) + 1, \text{ by the recurrence (1)} \\
&= 4 \cdot M(n-2) + 3, \text{ by arithmetic laws} \\
&= 8 \cdot M(n-3) + 7, \text{ by the recurrence (1) and arithm.} \\
&= 2^3 \cdot M(n-3) + (2^3 - 1), \text{ by arithmetic laws} \\
&= \cdots \\
&= 2^k \cdot M(n-k) + (2^k - 1), \text{ by generalisation } 3 \rightsquigarrow k \\
&= \cdots \\
&= 2^n \cdot M(0) + (2^n - 1), \text{ when } k = n \\
&= 2^n - 1, \text{ by the recurrence (1)}
\end{aligned}
$$

Hanoi: Proof by Induction

Theorem: $M(n) = 2^n - 1$, for all $n \geq 0$.

Proof:

Basis: If $n = 0$, then $M(n) = 0 = 2^0 - 1$, by (1).

Induction:
Assume the theorem holds for $n - 1$, for some $n > 0$. Then:

$$\begin{aligned} M(n) &= 2 \cdot M(n-1) + 1, \text{ by the recurrence (1)} \\ &= 2 \cdot (2^{n-1} - 1) + 1, \text{ by the assumption above} \\ &= 2^n - 1, \text{ by arithmetic laws} \quad \square \end{aligned}$$

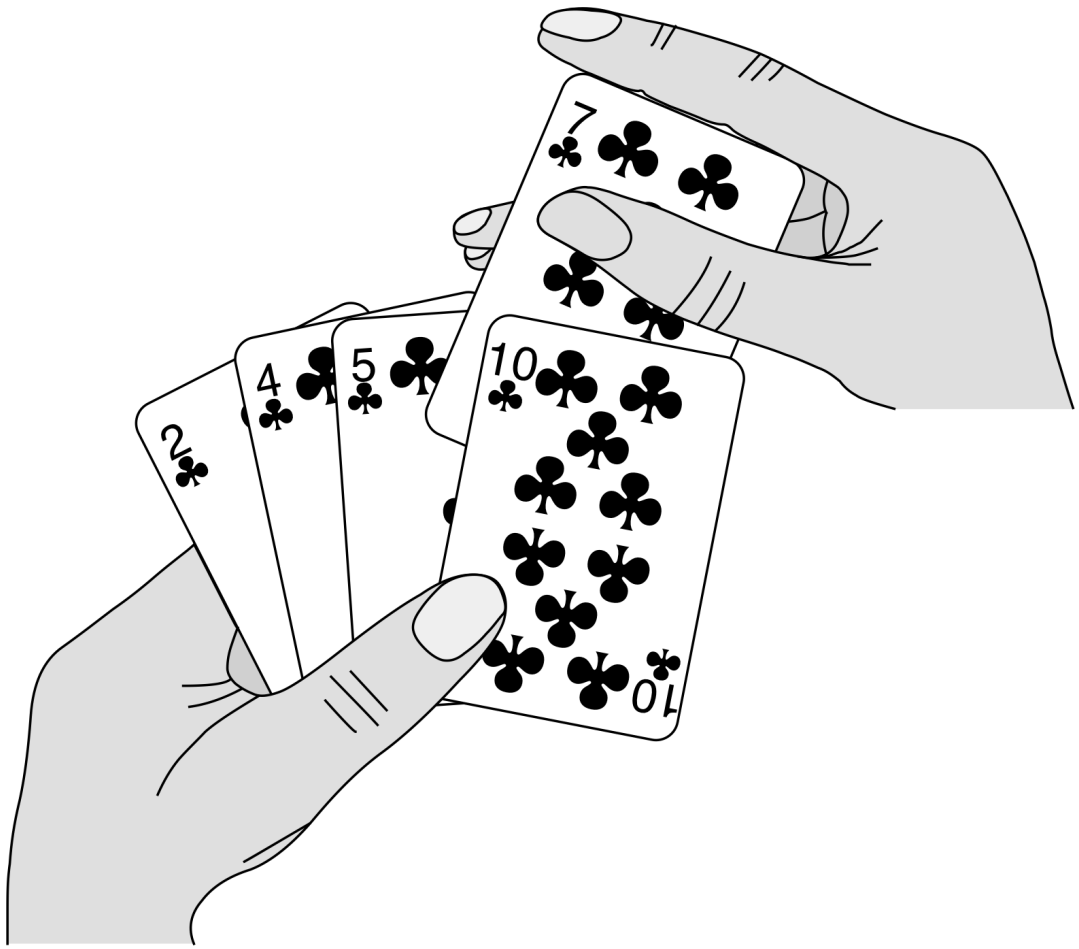
Hence: The move complexity of `hanoi(`n`,...)` is $\Theta(2^n)$.

Note that $2^{64} - 1 \approx 18.5 \cdot 10^{18}$ moves will take 580 billion years at 1 move / second, but the Big Bang is was only only about 14 billion years ago.

Insertion Sort

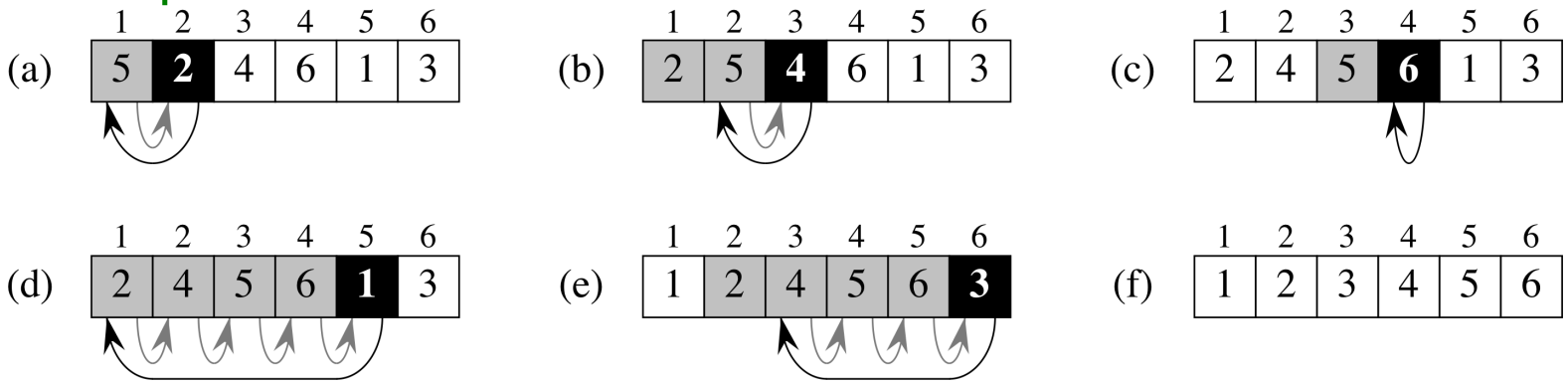
Assume we want to sort an array of n elements by non-decreasing order.

The idea of the insertion sort algorithm is the same as many people use when sorting a hand of playing cards:



Example and Invariant

Example



At any moment of insertion sorting, the array is divided into:

- The sorted section (here at the lower indices).
- The section not looked at yet.

Example

Before step (b) above:

2	5	4	6	1	3

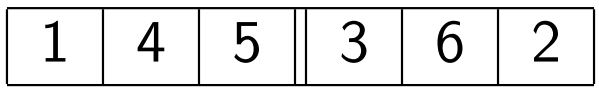
Algorithm

Initially place the dividing line between the first two elements (as a one-element array is always sorted).

While the dividing line is not after the last element:
Advance the dividing line one notch to the right, and insert the newly encountered element into the sorted section.

Example

Before:



After:

Analysis

Insertion sort is implemented by two functions:

- The main function, called `sort`, processes each element, inserting it into the sorted section.
- The help function, called `ins`, is called by `sort` to insert one element into the sorted section.

The amount of work done by `ins` depends on how many larger elements are in the sorted section:

- If none, then a single comparison is performed.
- If several, then each of them is compared and moved.
- At worst, every element is larger than the inserted one.

The runtime for the help function `ins`, denoted by T_{ins}, depends thus on the size i of the sorted section:

$$T_{\text{ins}}(i) = \Theta(i) \text{ at worst}$$

Analysis

The runtime for the main function `sort`, denoted by T_{isort}, is the sum of the runtimes of the help function `ins`:

$$T_{\text{isort}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T_{\text{ins}}(1) + T_{\text{ins}}(2) + \cdots + T_{\text{ins}}(n-1) & \text{if } n > 1 \end{cases}$$

where n is the number of elements.

Equivalently, and as a recurrence:

$$T_{\text{isort}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T_{\text{isort}}(n-1) + T_{\text{ins}}(n-1) & \text{if } n > 1 \end{cases} \tag{2}$$

Analysis

If $T_{\text{ins}}(i) = T_{\leq} = \Theta(1)$ for **all** i (the best case: the elements are initially already in sorted order), then:

$$T_{\text{isort}}(n) = (n-1) \cdot \Theta(1) = \Theta(n)$$

If $T_{\text{ins}}(i) = i \cdot (T_{\leq} + T_{\text{move}}) = i \cdot \Theta(1)$ for **all** i (the worst case: the elements are initially in reverse-sorted order), then:

$$T_{\text{isort}}(n) = \sum_{j=1}^{n-1} (j \cdot \Theta(1)) = \Theta(1) \cdot \frac{n \cdot (n-1)}{2} = \Theta(n^2)$$

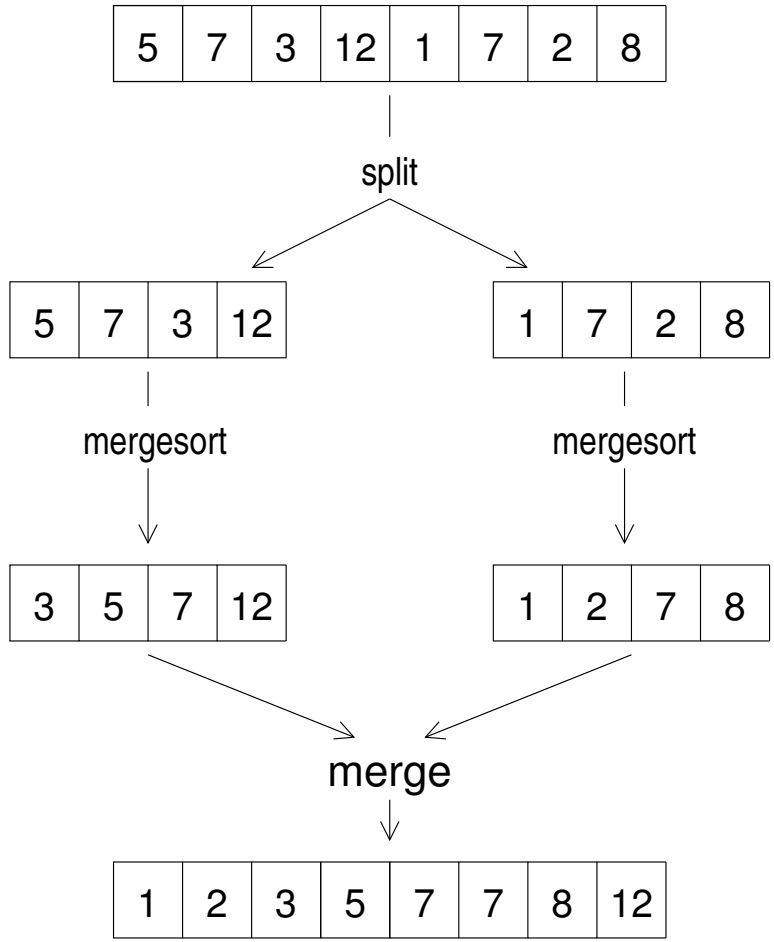
If $T_{\text{ins}}(i) = \frac{i}{2} \cdot (T_{\leq} + T_{\text{move}}) = i \cdot \Theta(1)$ on **average** for all i (the average case: the elements are moved on average half-way into the sorted section), then:

$$T_{\text{isort}}(n) = \Theta(n^2)$$

Overall, we say that insertion sort runs in $\Theta(n)$ time at best, and in $\Theta(n^2)$ time on average and at worst.

Merge Sort (John von Neumann, 1945)

Runtime: Always $\Theta(n \cdot \log n)$ for n elements.
Apply the divide & conquer (& combine) principle:



Splitting a List Into Two 'Halves'

The order of the elements inside `A` and `B` is irrelevant!

Naïve program:

```
fun split L =
    let val t = (length L) div 2
    in  (List.take (L, t), List.drop (L, t)) end
```

`split` L always takes $|L| + 2 \cdot \left\lfloor \frac{|L|}{2} \right\rfloor = \Theta(|L|)$ time.

Merging Two Sorted Lists

```
merge (L, M)
EXAMPLE: merge ([3,5,7,12],[1,2,7,8,13]) = [1,2,3,5,7,7,8,12,13]

fun merge ([], M) = M
  | merge (L, []) = L
  | merge (L as x::xs, M as y::ys) =
    if x > y then
        y :: merge (L, ys)
    else
        x :: merge (xs, M)
```

Merge Sort Program

```
sort L

ALGORITHM: merge sort
fun sort [] = []
  | sort [x] = [x]
  | sort xs =
    let
        val (ys, zs) = split xs
    in
        merge (sort ys, sort zs)
    end
```

Analysis — Base Case

$T_{\texttt{msort}}(n)$ is the time of running `sort` on n elements:

Base cases: $(n \leq 1)$:

Constructing a list of 0 or 1 element takes $\Theta(1)$ time.

Analysis — Recursive Case

Recursive case: $(n > 1)$:

- Divide: `split` xs takes $\Theta(|xs|) = \Theta(n)$ time, by split.
- Conquer: The recursive calls `sort` ys and `sort` zs take $T_{\texttt{msort}}\left(\frac{n}{2}\right)$ time each, because $|ys| + |zs| = n$ and $||ys| - |zs|| \leq 1$, by the post-condition of `split`.
  (If n is odd, then $\frac{n}{2}$ is not an integer, but this does not matter asymptotically.)
- Combine: `merge(sort` ys`, sort` zs`)` takes $\Theta(n)$ time, by merge, since $|\texttt{sort}\ L| = |L|$
  (by the post-condition of `sort`)
  and thus $|\texttt{sort}\ ys| + |\texttt{sort}\ zs| = |ys| + |zs| = n$
  (by the post-condition of `split`).

Analysis (continued)

Hence the runtime recurrence:

$$T_{\texttt{msort}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ \Theta(n) + 2 \cdot T_{\texttt{msort}}(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

which simplifies into:

$$T_{\texttt{msort}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T_{\texttt{msort}}(n/2) + \Theta(n) & \text{if } n > 1 \end{cases} \tag{3}$$

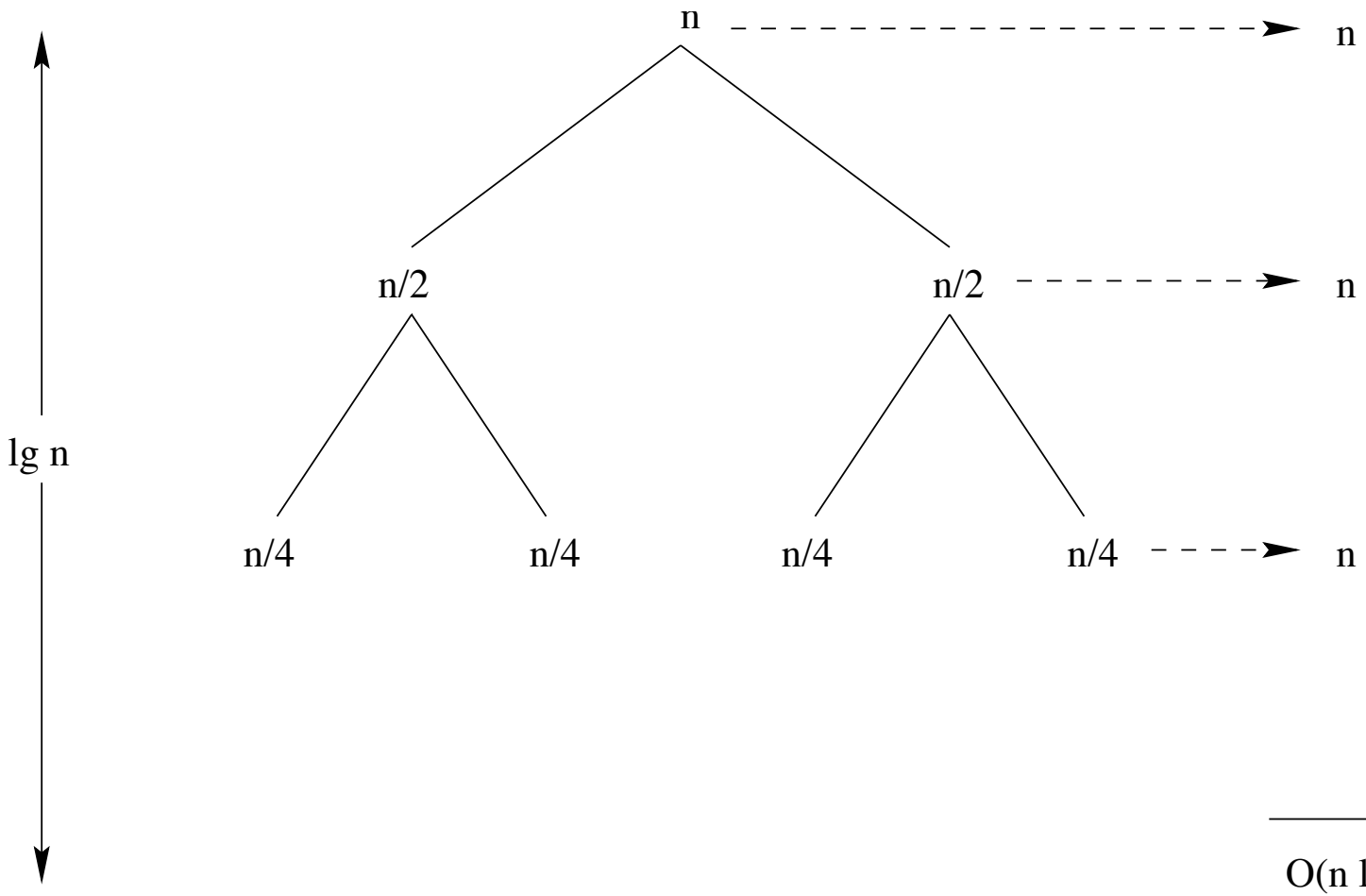
where $\Theta(n)$ is the total time of dividing and combining.

The closed form is $T_{\texttt{msort}}(n) = \Theta(n \cdot \log n)$, in all cases. We'll see later why this is true.

Merge sort is better than insertion sort in the average and worst cases; insertion sort is better for nearly-sorted data.

The Recursion-Tree Method

You can visualise the running of recursive algorithm as a tree (often called a recursion tree). The recursion tree for the merge sort recurrence is:



Elementary Mathematics Revision

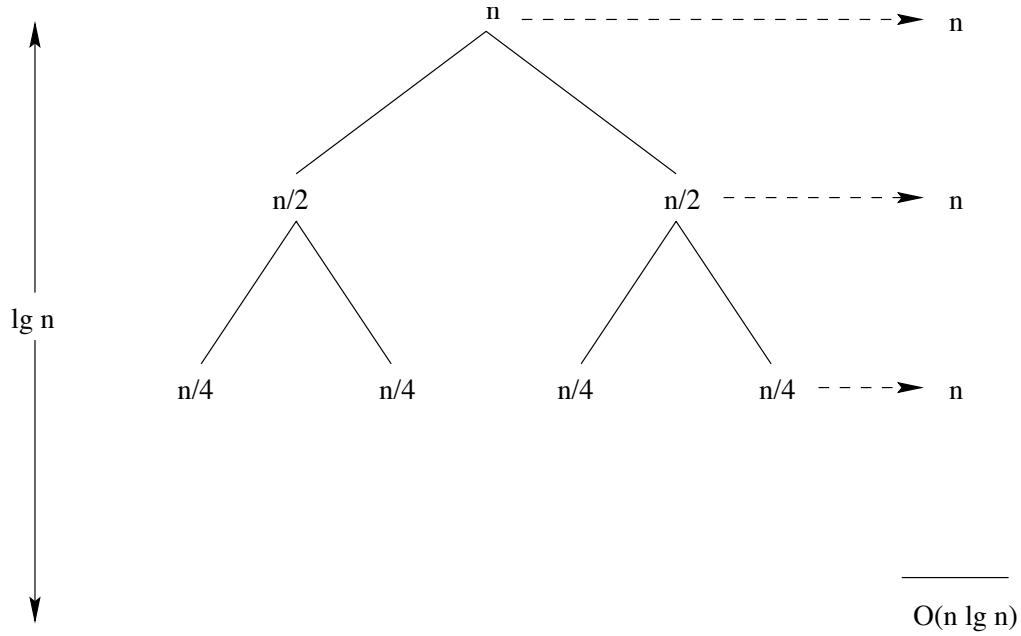
$$\log_2 x = y \Leftrightarrow 2^y = x$$

Hence

$$\log_2 2^k = k$$

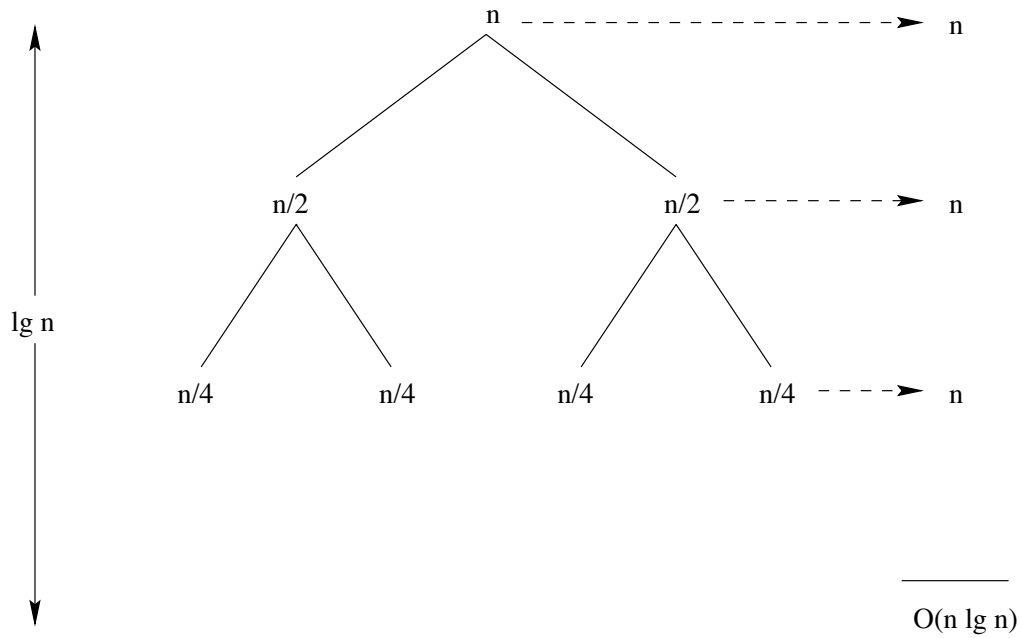
This is why if you keep dividing a number n by 2 you will get to a number close to 1 in approximately $\log_2 n$ steps. This is the key to a lot of algorithm analysis and clever algorithms that run in $nlog_2 n$ steps.

The Recursion-Tree Method



Why does the tree have height $\log_2(n)$?
Why does each level sum to n?

The Recursion-Tree Method



The total complexity is the height $\log_2(n)$ times the amount of work done at each level n, this gives the $n \log_2(n)$ bound for sorting.

Types of Recurrences

We have already observed that a recurrence of the form

- $T(n) = T(n-1) + \Theta(1)$ gives $\Theta(n)$
- $T(n) = T(n-1) + \Theta(n)$ gives $\Theta(n^2)$

Types of Recurrences

Divide-and-conquer algorithms give recurrences of the form

$$T(n) = a \cdot T(n/b) + f(n)$$

where a sub-problems are produced, each of size n/b, and $f(n)$ is the total time of dividing the input and combining the sub-results.

The recursion tree for an algorithm gives an idea of how much work is done. It is all a question of the relationship between the work done at each level $f(n)$ and how many sub problems you have. If $f(n)$ is too big then it dominates the complexity. The relationship is captured by the master theorem (see later).

Proof of the Merge-Sort Recurrence

This proof gives the general flavour of solving divide-and-conquer recurrences.
The formal proof is complicated by technical details, such as when n is not an integer power of b.
We ignore such issues in this proof sketch.

Theorem: If (compare with recurrence (3) three pages ago)

$$T(n) = \begin{cases} 2 & \text{if } n = 2 = 2^k \text{ for } k = 1 \\ 2 \cdot T(n/2) + \Theta(n) & \text{if } n = 2^k \text{ for } k > 1 \end{cases} \tag{4}$$

then $T(n) = \Theta(n \cdot \log n)$, for all $n = 2^k$ with $k \geq 1$.

Proof by Induction

Proof: For $n = 2^k$ with $k \geq 1$, the closed form $T(n) = \Theta(n \cdot \log n)$ becomes $T(2^k) = 2^k \cdot \log 2^k$.

Basis: If $k = 1$ (and hence $n = 2$), then $T(n) = 2 = 2 \cdot \log 2$.

Induction: Assume the theorem holds for some $k \geq 1$. Then:

$$\begin{aligned}
T(2^{k+1}) &= 2 \cdot T(2^{k+1}/2) + 2^{k+1}, \text{ by the recurrence (4)} \\
&= 2 \cdot T(2^k) + 2^{k+1}, \text{ by arithmetic laws} \\
&= 2 \cdot (2^k \cdot \log 2^k) + 2^{k+1}, \text{ by the assumption} \\
&= 2^{k+1} \cdot (\log 2^k + 1), \text{ by arithmetic laws} \\
&= 2^{k+1} \cdot (\log 2^k + \log 2^1), \text{ by arithmetic laws} \\
&= 2^{k+1} \cdot \log 2^{k+1}, \text{ by arithmetic laws} \quad \square
\end{aligned}$$

The Master Method and Master Theorem

The closed form for a recurrence $T(n) = a \cdot T(n/b) + f(n)$ reflects the battle between the two terms in the sum. Think of $a \cdot T(n/b)$ as the process of "distributing the work out" to $f(n)$, where the actual work is done.

Theorem :

1. If $f(n)$ is dominated by $n^{\log_b a}$ (see the next page), then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n)$ and $n^{\log_b a}$ are balanced (if $f(n) = \Theta(n^{log_b a})$), then $T(n) = \Theta(n^{log_b a} \cdot \lg n)$.
3. If $f(n)$ dominates $n^{\log_b a}$ and if the regularity condition (see the next page) holds, then $T(n) = \Theta(f(n))$.

Dominance and the Regularity Condition

The three cases of the Master Theorem depend on comparing $f(n)$ to $n^{\log_b a}$. However, it is not sufficient for $f(n)$ to be "just a bit" smaller or bigger than $n^{\log_b a}$. Cases 1 and 3 only apply when there is a polynomial difference between these functions, that is when the ratio between the dominator and the dominee is asymptotically larger than the polynomial n^{ϵ} for some constant $\epsilon > 0$.

Dominance and the Regularity Condition

Example: n^2 is polynomially larger than both $n^{1.5}$ and $\lg n$.

Counter-Example: $n \cdot \lg n$ is not polynomially larger than n.

In Case 3, a regularity condition requires $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n.
(All the f functions in this course will satisfy this condition.)

Proof of the Master Theorem

The proof is quite involved and technical.

- In case 2 the tree and the work is balanced the height is given by $n^{\log_b a}$, and there is not too much work combining the sub-problems. Further, you are splitting enough to not to do too much work in each sub-branch. In merge sort we divided into $k/2$ parts.
- In case 1 when you split, there is not much work in combining the sub-problems, and so the complexity is dominated by the height of the tree.
- In case 3 the combining of the sub-problems dominates the work.

Gaps in the Master Theorem

The Master Theorem does not cover all possible recurrences of the form $T(n) = a \cdot T(n/b) + f(n)$:

- Cases 1 and 3: The difference between $f(n)$ and $n^{log_b a}$ might not be polynomial.
  Counter-Example: The Master Theorem does not apply to the recurrence $T(n) = 2 \cdot T(n/2) + n \cdot \lg n$, despite it having the proper form. We have $a = 2 = b$, so we need to compare $f(n) = n \cdot \lg n$ to $n^{\log_b a} = n^1 = n$. Clearly, $f(n) = n \cdot \lg n > n$ for large enough n, but the ratio $f(n)/n$ is $\lg n$, which is asymptotically less than the polynomial n^ϵ for any constant $\epsilon > 0$, so we are not in Case 3.
- Case 3: The regularity condition might not hold.

Common Cases of the Master Theorem

a	b	$n^{\log_b a}$	$f(n)$	Case	$T(n)$
1	2	n^0	$\Theta(1)$	2	$\Theta(\lg n)$
			$\Theta(\lg n)$	none	$\Theta(?)$
			$\Theta(n \cdot \lg n)$	3	$\Theta(n \cdot \lg n)$
			$\Theta(n^k)$, with $k > 0$	3	$\Theta(n^k)$
2	2	n^1	$\Theta(1)$	1	$\Theta(n)$
			$\Theta(\lg n)$	1	$\Theta(n)$
			$\Theta(n)$	2	$\Theta(n \cdot \lg n)$
			$\Theta(n \cdot \lg n)$	none	$\Theta(?)$
			$\Theta(n^k)$, with $k > 1$	3	$\Theta(n^k)$

What is n?

Complexity theory is a complex and fascinating subject.

We have and will writte expressions such as $T(n)$:

- Sorting a list of n elements.
- Given a graph with n nodes find the shortest path.

The Tower of Hannoi example is a bit complicated, so don't think about it here.

What is n?

Complexity theory is formally defined in terms of an abstract computer that uses a tape as a memory. This is called a Turing machine.

The question you need to ask yourself, how many bits or bytes do I need to represent something?

What is n?

So a list of n elements will need $n \cdot w$ bytes where w is the word size of your computer.

A graph of n nodes might be represented by a $n \cdot n$ table of single bits.

Assumption in complexity theory all equivalent representations differ by a constant factor. A number could be represented in binary, or unary or whatever. In big-O analysis constant factors go away.

What is n?

Always think about the input size. Your time complexity is in terms of the input size.

Constant factors go away. So a list of length n takes $n \cdot w$ bytes, but you state the complexity in terms of n.

Be careful with the parameters of your algorithm. Try to understand how the affect the input size. This will become important later when we look at dynamic programming.

What is n?

You can assume that the word size is always big enough to handle the numbers that you need. This is because we can just increase the word size (on our abstract computer) to accommodate the numbers that we need to handle.

Thus you can assume arithmetic operations take constant time.

If this never worried you, then don't worry. If it worries you and it makes your head hurt, then don't worry and just assume things take constant time. If it has worried you, and you have worked out why, then be happy.