北京航空航天大學
BEIHANG UNIVERSITY

Introduction to Computer Science and Programming

Lecture 13

Sebastian Wandelt

Beihang University

Outline

- Recap
- Some “leftovers”
- Hardness of Problems
- Summary

Some leftovers

Heaps and Heapsort

Heaps and Heapsort

- Combines the better attributes of merge sort and insertion sort.
  - What could that be?

Heaps and Heapsort

- Combines the better attributes of merge sort and insertion sort.
  - Like merge sort, but unlike insertion sort
    - Running time is O(n log n).
  - Like insertion sort, but unlike merge sort:
    - Sorts in place.
- Makes use of a data structure (=heap) to manage information during the execution of an algorithm.
- The heap has many other applications beside sorting

Heap

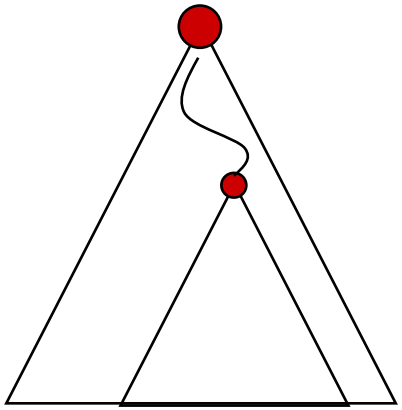
- A heap is a binary tree with a special property
- That sounds similar to …?
  - Which other special tree-property do you remember from previous lectures?

Heap

- A heap is a binary tree with a special property
- That sounds similar to …?
  - Which other special tree-property do you remember from previous lectures?
    - Binary search tree:
      - For each node:
        - » Left descendants smaller than node
        - » Right descendants larger than node
    - Balanced tree
      - For each node:
        - » Number of left children is equal to number of right children +-1

Heap Property (Max vs. Min)

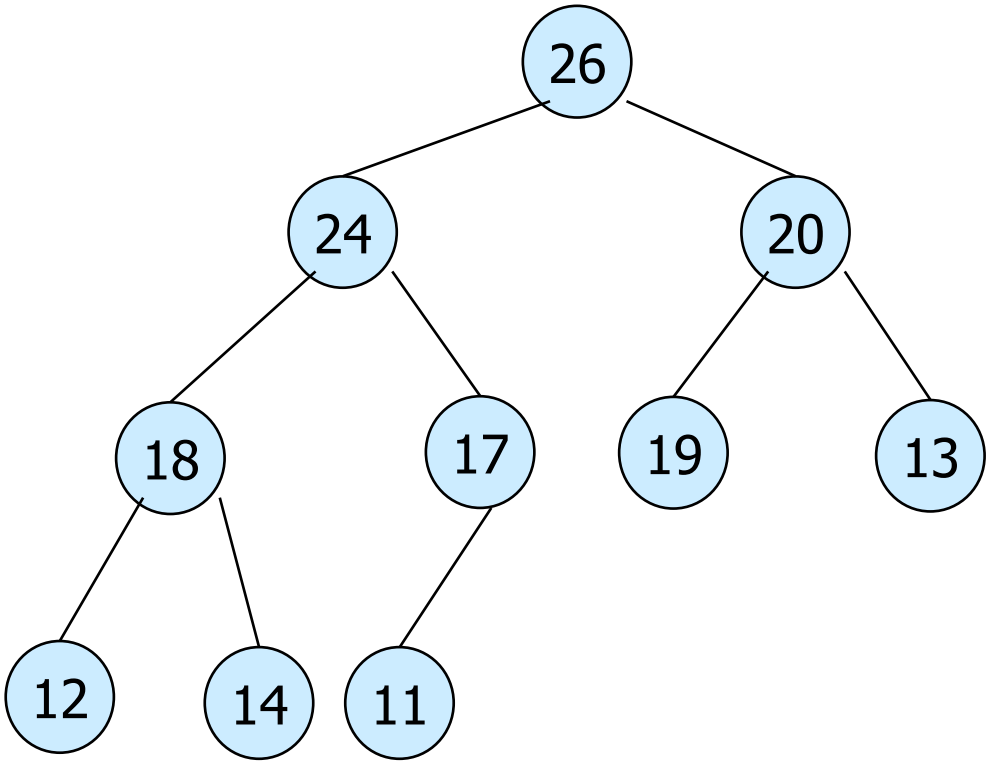
- Max-Heap
  - For every node excluding the root, value is at most that of its parent
  - Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at subtree root.



- Min-Heap
  - Inverse …

Max heap: Example

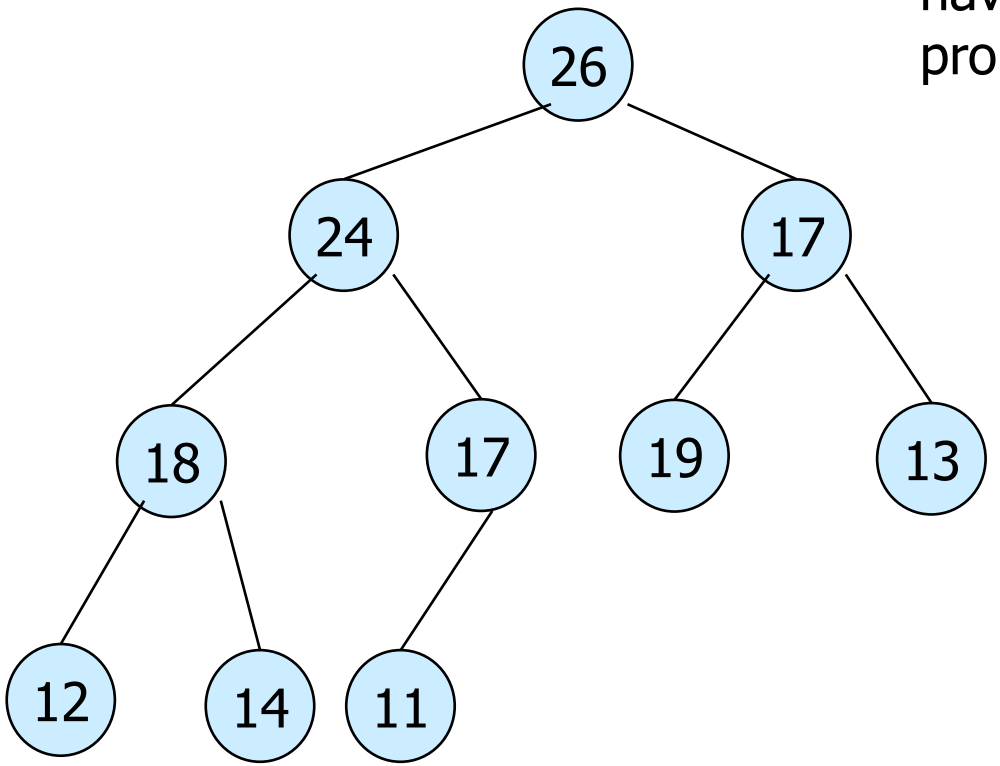
- For every node excluding the root, value is at most that of its parent:



Max heap: Another Example?

- For every node excluding the root, value is at most that of its parent:

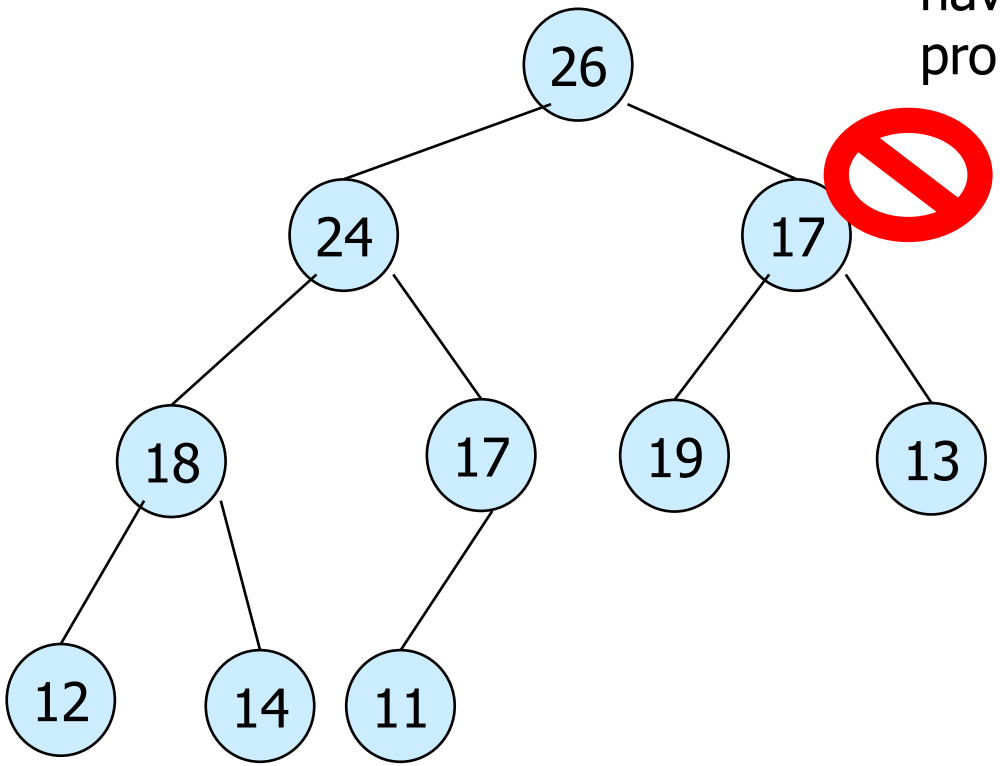
Does this tree have the heap property?



Max heap: Another Example?

- For every node excluding the root, value is at most that of its parent:

Does this tree have the heap property?



Heaps in Sorting

- Use max-heaps for sorting
- Create a max-heap for the input list
- Sorting algorithm (executed until the heap is empty):
  - We can always find the maximum element in a heap easily
  - After extracting the maximum, we need to do some kind of "repair" to our tree

Can we still avoid dealing with trees?

Data Structure Binary Heap

- List/Array viewed as a nearly complete binary tree.
  - Physically – linear array.
  - Logically – binary tree, filled on all levels (except lowest.)
- Map from array elements to tree nodes and vice versa
  - Root – $A[0]$
  - Left[i] – $A[2i+1]$
  - Right[i] – $A[2i+2]$
  - Parent[i] – $A[\lfloor i/2 \rfloor]$
- length[A] – number of elements in array A.
- heap-size[A] – number of elements in heap stored in A.
  - heap-size[A] $\leq$ length[A]

Heaps – Example

26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10

Max-heap as an array.

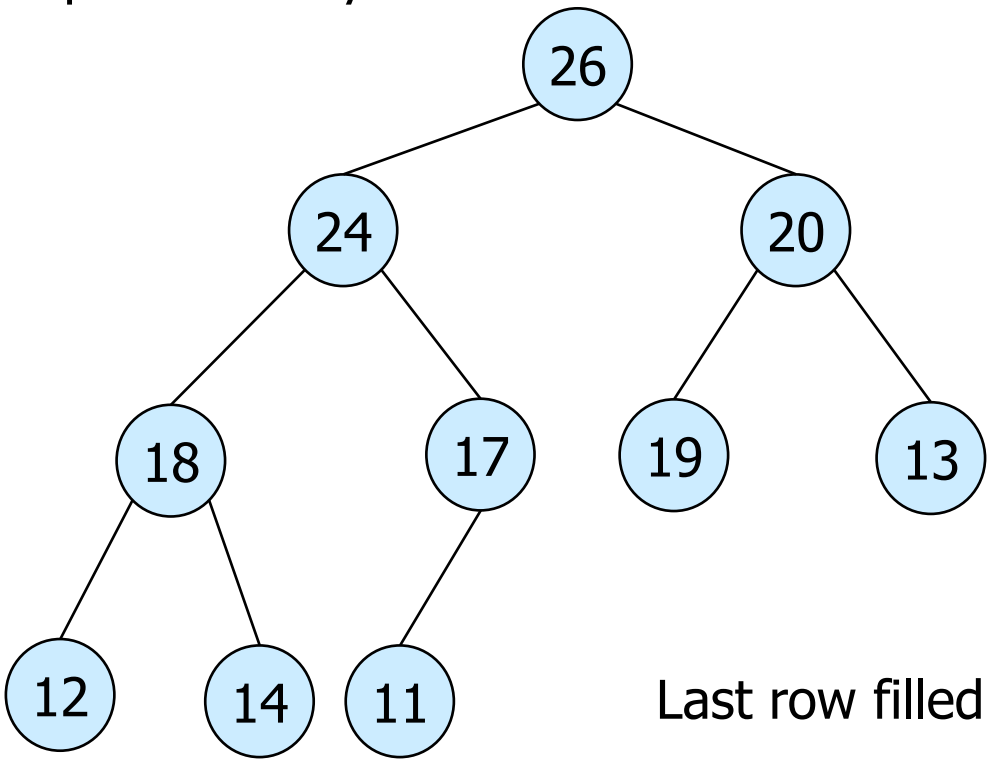
What is the corresponding max-heap tree?

Heaps – Example

26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10

Max-heap as an array.

Max-heap as a binary tree.



Last row filled from left to right.

Heaps in Sorting

- Again, how exactly can we use max-heaps for sorting?

Heaps in Sorting

- Use max-heaps for sorting.
- The array representation of max-heap is not sorted!
- Steps in sorting
  1. Convert the given array of size *n* to a max-heap (*BuildMaxHeap*)
  2. Swap the first and last elements of the array.
     - Now, the largest element is in the last position – where it belongs.
     - That leaves *n* – 1 elements to be placed in their appropriate locations.
     - However, the array of first *n* – 1 elements is no longer a max-heap.
     - Float the element at the root down one of its subtrees so that the array remains a max-heap
     - Repeat step 2 until the array is sorted.

Heaps in Sorting

- Please try this on the following example:
  - [2,6,3,5,4,7,1]

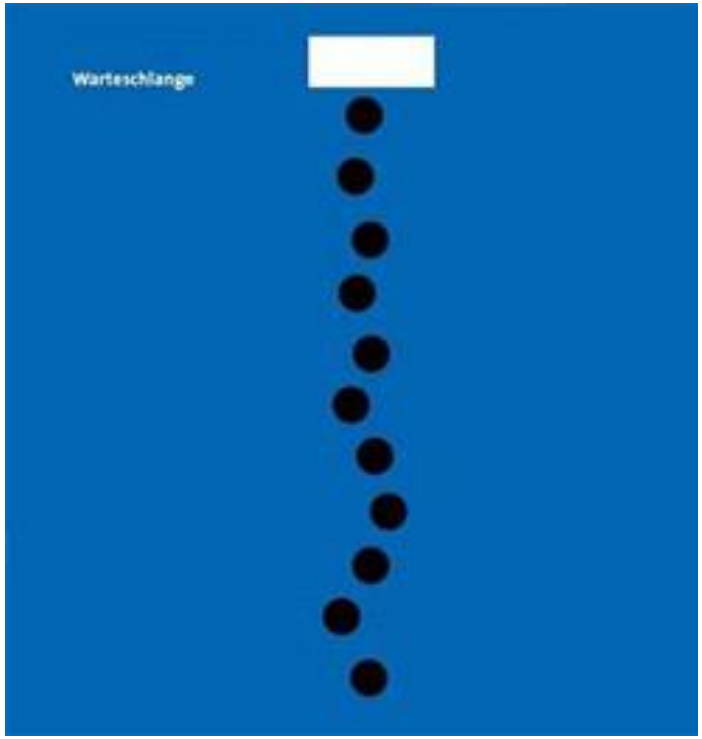
Stacks and Queues

Stacks and Queues

- Both data structures are dynamic collections, in which the order of element removal is **prespecified**

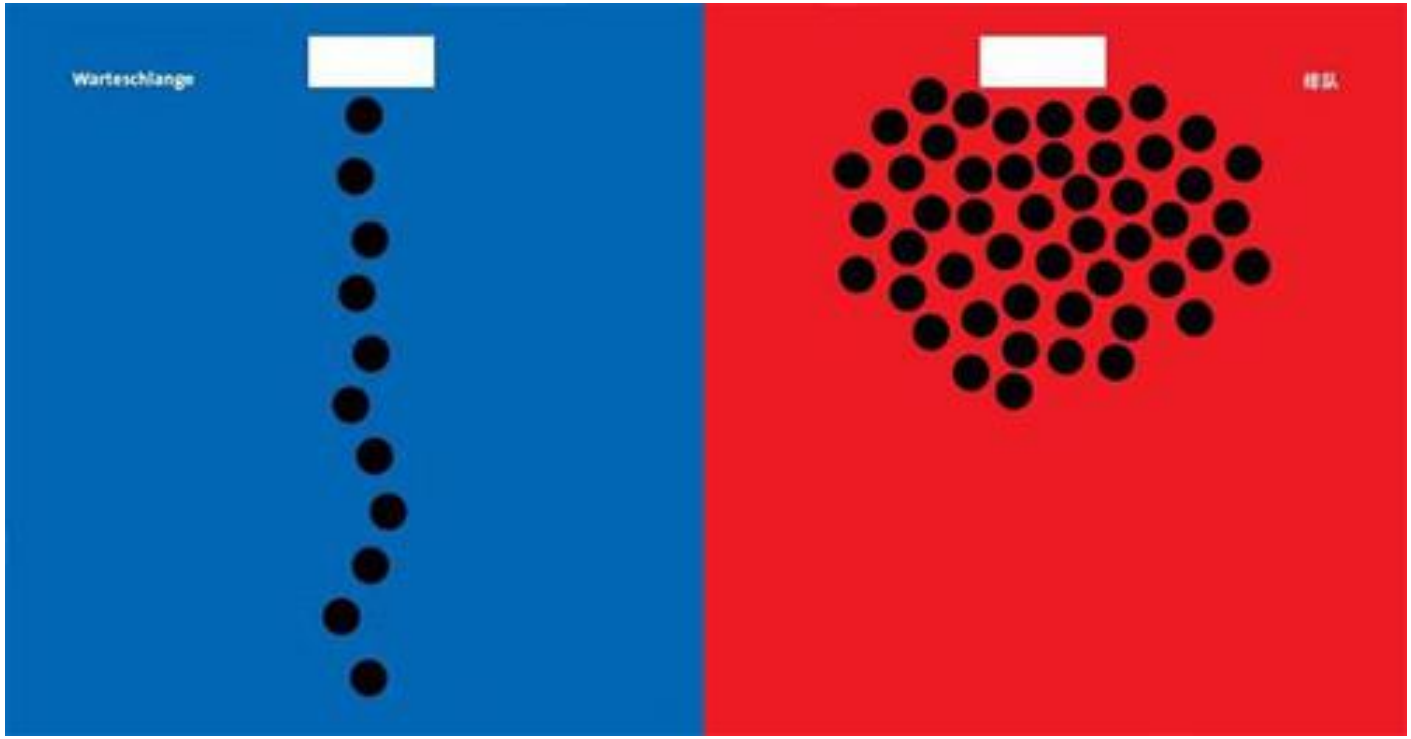
Stacks and Queues

- Both data structures are dynamic collections, in which the order of element removal is **prespecified**
- Motivation:



Stacks and Queues

- Both data structures are dynamic collections, in which the order of element removal is **prespecified**
- Motivation:



Stacks and Queues

- Both data structures are dynamic collections, in which the order of element removal is **prespecified**

- **Two operations on queues:**
  - ***enqueue*(x) … add an element x to the end of the collection**
  - **x=dequeue() … retrieve and remove the *oldest* element**
  - **FIFO (First In First Out)**
- Two operations on stacks:
  - push(x) … add an element x to the top of the collection
  - x=pop() … retrieve and remove the most recently added element
  - LIFO (Last In First Out)

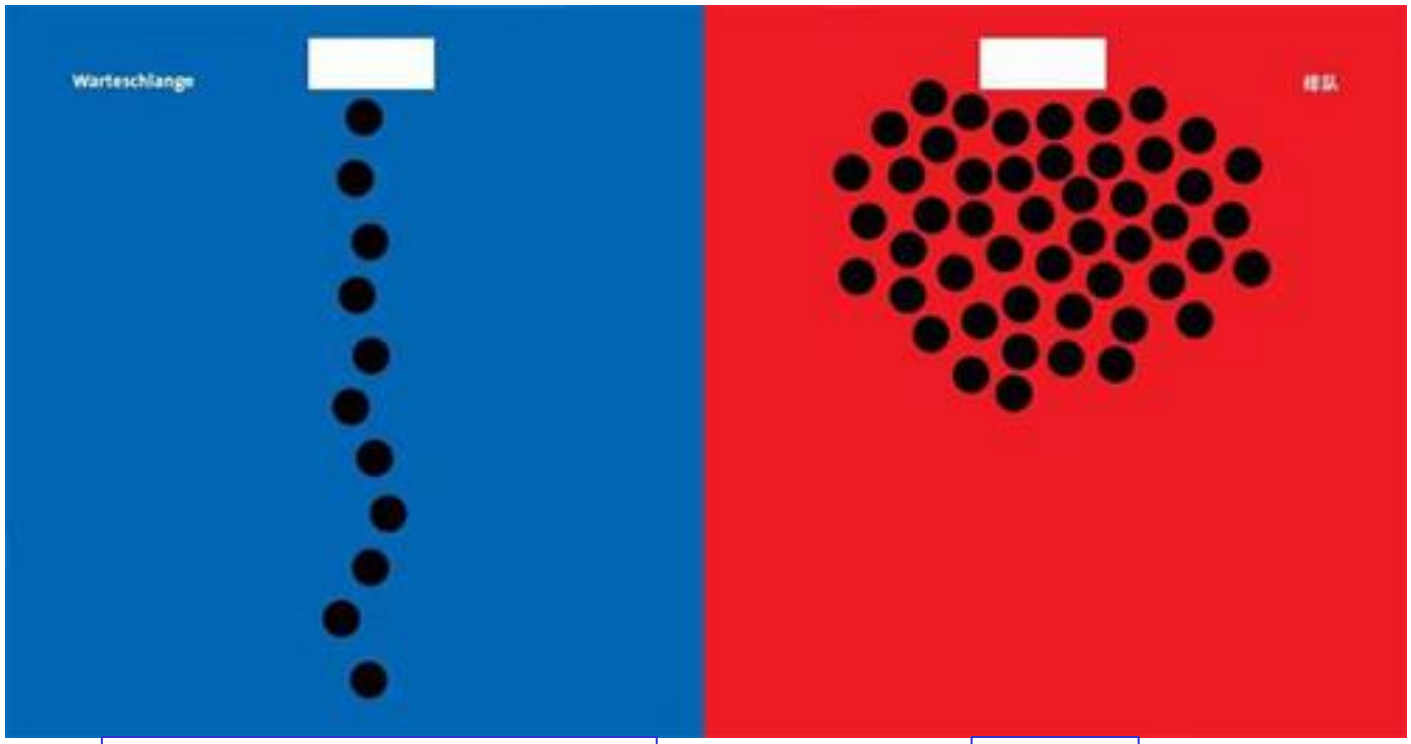
Stacks and Queues

- Both data structures are dynamic collections, in which the order of element removal is **prespecified**

- Two operations on queues:
  - *enqueue*(x) … add an element x to the end of the collection
  - x=dequeue() … retrieve and remove the *oldest* element
  - FIFO (First In First Out)
- **Two operations on stacks:**
  - **push(x) … add an element x to the top of the collection**
  - **x=pop() … retrieve and remove the most recently added element**
  - **LIFO (Last In First Out)**

Stacks and Queues

- Both data structures are dynamic collections, in which the order of element removal is **prespecified**
- Motivation:

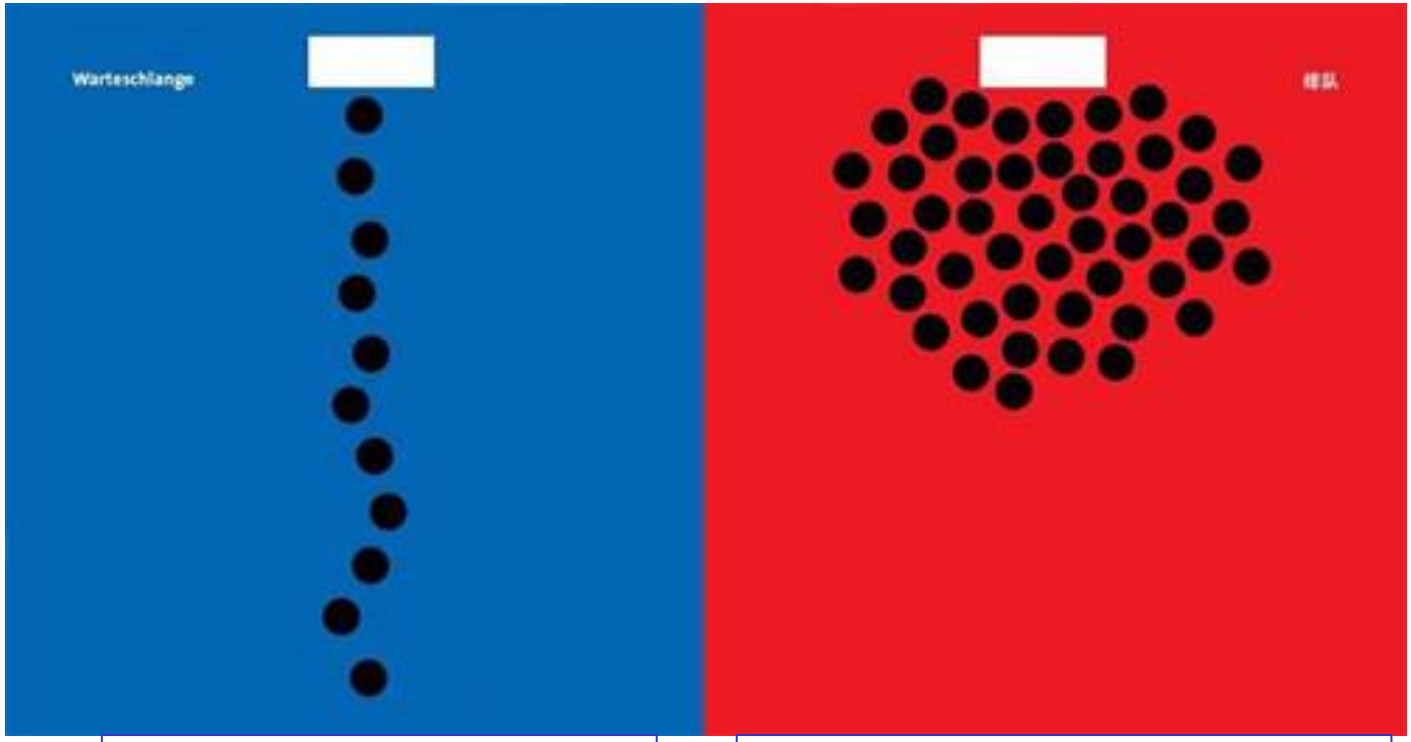


FIFO (Queue)

???

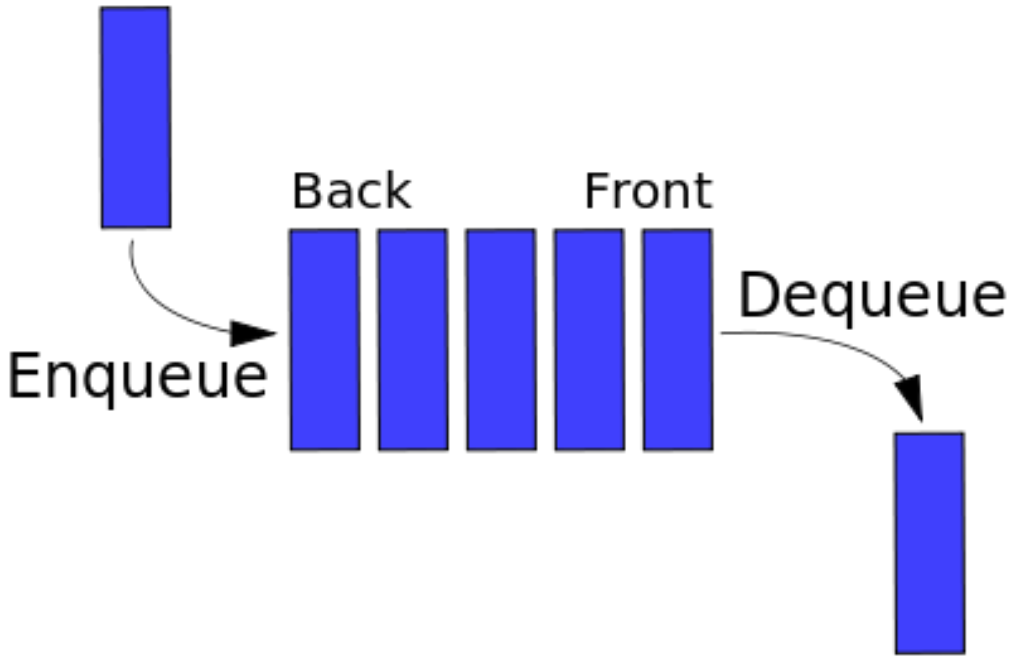
Stacks and Queues

- Both data structures are dynamic collections, in which the order of element removal is **prespecified**
- Motivation:



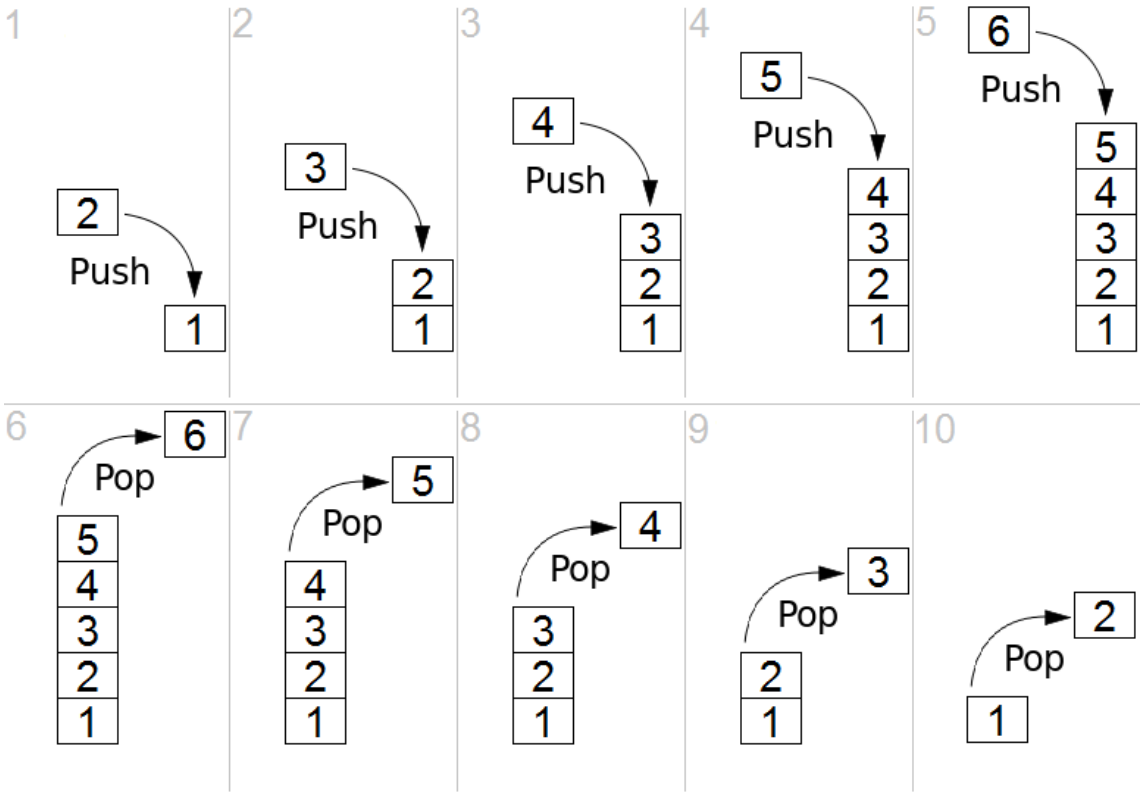
FIFO (Queue) **LIFO? Not really**

Queue (FIFO)



- Priority management
- Many real-world examples
  - Transportation, Administration, …

Stack (LIFO)



- Real-world example are harder to find than for Queues
  - Yet, our brain/memory somehow is a vague stack.
- Stacks are mainly used in algorithms!

Stacks and Queues

- Both can be implemented by using lists easily
  - How?

Hardness of Problems

Subset-sum problem

- Given a set of integers, does any non-empty subset of them add up to a given number k?
- How would Python code for that function look like?

3-partition problem

- Given a list of integers, can S be partitioned into triples, such that the sum of each triple is equal?
- [20, 23, 25, 45, 27, 40] can be partitioned into the two triples:
  - [20, 25, 45] …sum=90
  - [23, 27, 40] …sum=90

- How would Python code for that function look like?

3-partition problem: Solution

- Try at home ☺

Complexity classes

Function	Common Name
N!	factorial
2^N	Exponential
N^d, d > 3	Polynomial
N^3	Cubic
N^2	Quadratic
$N\sqrt{N}$	N Square root N
N log N	N log N
N	Linear
$\sqrt{N}$	Square root of n
log N	Logarithmic
1	Constant

Complexity classes

Examples?

Function	Common Name
N!	factorial
2^N	Exponential
N^d, d > 3	Polynomial
N^3	Cubic
N^2	Quadratic
$N\sqrt{N}$	N Square root N
N log N	N log N
N	Linear
$\sqrt{N}$	Square root of n
log N	Logarithmic
1	Constant

Complexity classes

	Function	Common Name
Most naïve Sort	N!	factorial
Naïve Subsetsum	2^N	Exponential
	N^d, d > 3	Polynomial
	N^3	Cubic
Bubble Sort	N^2	Quadratic
	$N\sqrt{N}$	N Square root N
Merge Sort	N log N	N log N
Naïve search in List	N	Linear
	$\sqrt{N}$	Square root of n
Balanced BST search	log N	Logarithmic
Array index lookup	1	Constant

Complexity classes

Which complexity class is acceptable?

Function	Common Name
N!	factorial
2^N	Exponential
N^d, d > 3	Polynomial
N^3	Cubic
N^2	Quadratic
$N\sqrt{N}$	N Square root N
N log N	N log N
N	Linear
$\sqrt{N}$	Square root of n
log N	Logarithmic
1	Constant

Complexity classes – Run times

Which complexity class is acceptable?

Depends on problem and input size!

Assume N = 100,000 and processor speed is 1*10**9 operations per second

Function	Running Time
2^N	3.2 x $10^{30,086}$ years
N^4	3171 years
N^3	11.6 days
N^2	10 seconds
$N\sqrt{N}$	0.032 seconds
N log N	0.0017 seconds
N	0.0001 seconds
$\sqrt{N}$	3.2 x 10^{-7} seconds
log N	1.2 x 10^{-8} seconds

What else ....?

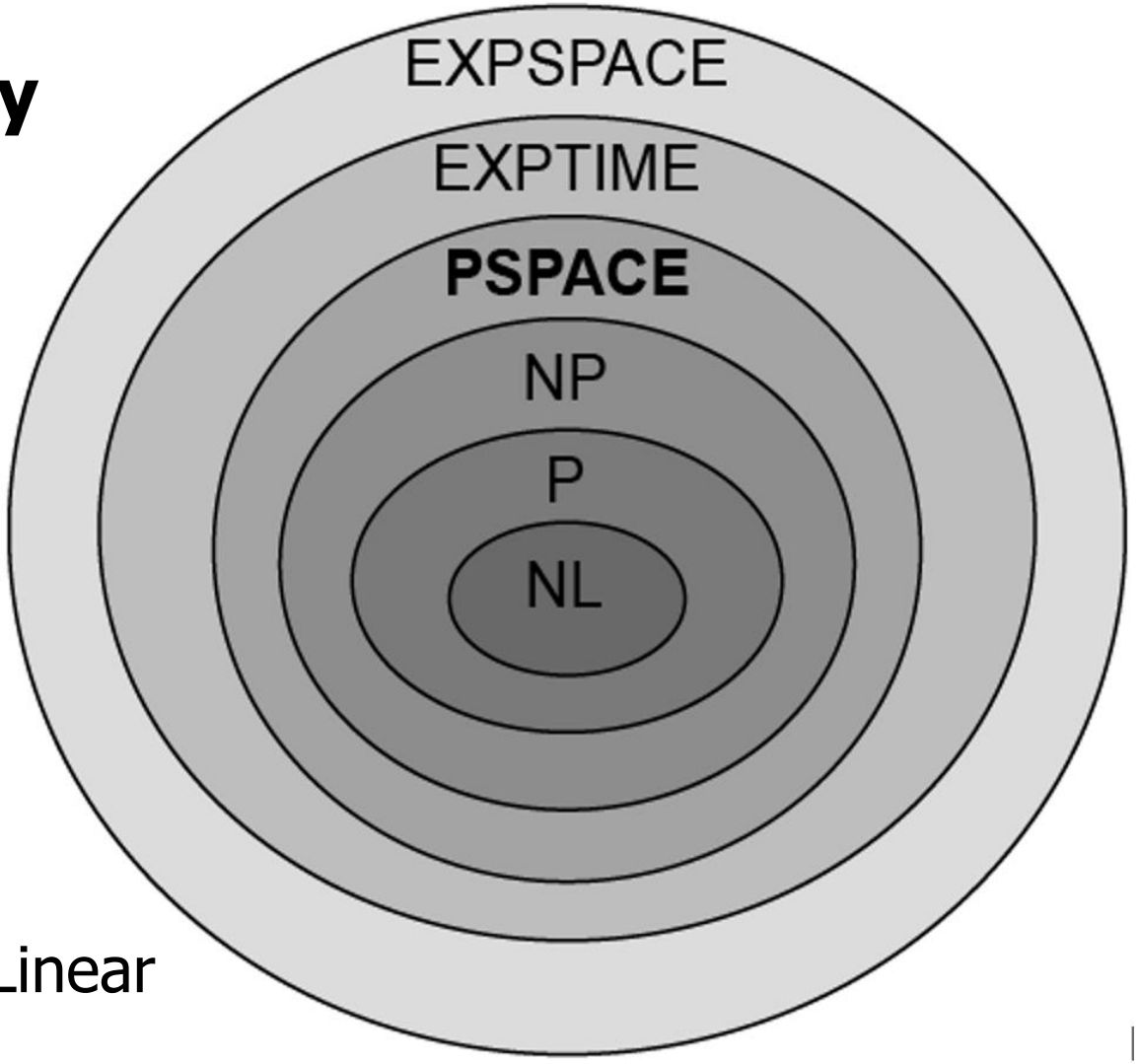
Please note ...

- The next few slides are just a very rough overview
- Not part of the exam anymore

~~Please note ...~~

- ~~The next few slides are just a very rough overview~~
- ~~Not part of the exam anymore~~

Complexity Theory



- NL=Non-Deterministic Linear
- P=Polynomial
- NP= Non-Deterministic Polynomial
- EXPTIME=Exponential Time

What about this Non-Determinism?

- Computers are - by design – deterministic
- One function can be executed at a time!
  - Any objection?

What about this Non-Determinism?

- Computers are - by design – deterministic
- One function can be executed at a time!
  - How about multiple processors?
  - How about multi-core machines?

What about this Non-Determinism?

- Computers are - by design – deterministic
- One function can be executed at a time!
  - How about multiple processors?
  - How about multi-core machines?
    - The speed–up obtained by parallelizing with multiple cores/processors is constant (with large input n)!
    - It does not change the complexity class at all!

What about this Non-Determinism?

- Computers are - by design – deterministic
- One function can be executed at a time!
  - How about multiple processors?
  - How about multi-core machines?
    - The speed–up obtained by parallelizing with multiple cores/processors is constant (with large input n)!
    - It does not change the complexity class at all!
- Suppose there exists a machine that can execute ANY number of functions in parallel
  - Assume that a single function call takes polynomial (P) time
  - Overall running time?

What about this Non-Determinism?

- Computers are - by design – deterministic
- One function can be executed at a time!
  - How about multiple processors?
  - How about multi-core machines?
    - The speed–up obtained by parallelizing with multiple cores/processors is constant (with large input n)!
    - It does not change the complexity class at all!
- Suppose there exists a machine that can execute ANY number of functions in parallel
  - Assume that a single function call takes polynomial (P) time
  - Overall running time?
    - Still polynomial time!

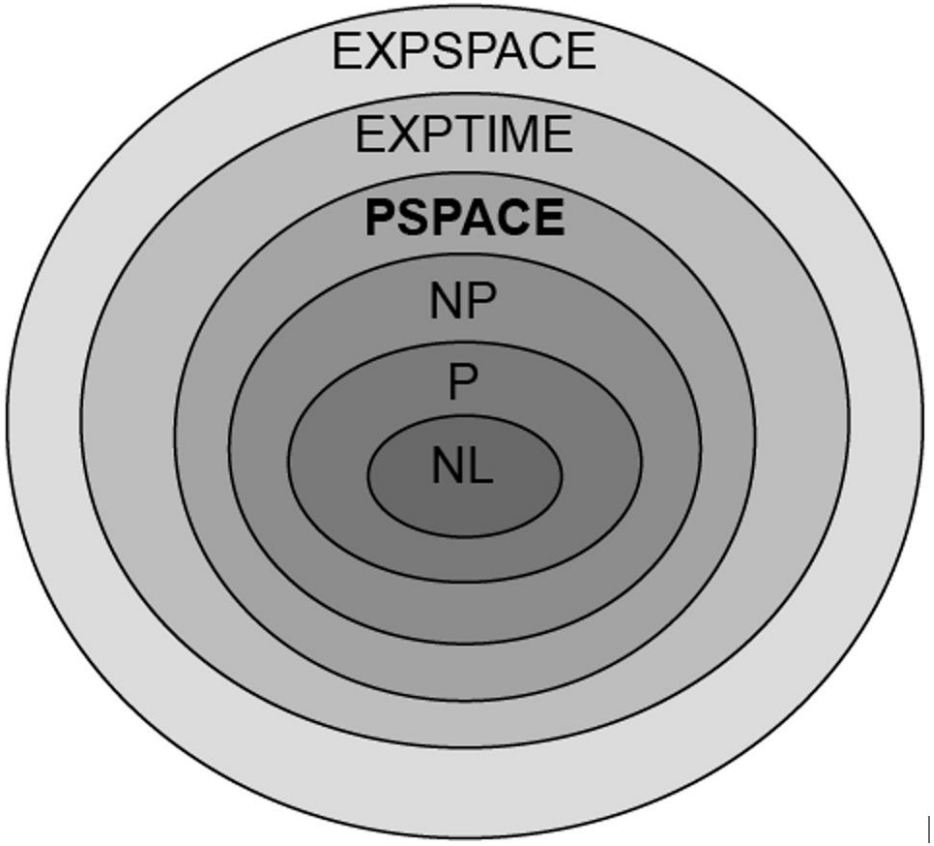
What about this Non-Determinism?

- Computers are - by design – deterministic
- One function can be executed at a time!
  - How about multiple processors?
  - How about multi-core machines?
    - The speed–up obtained by parallelizing with multiple cores/processors is constant (with large input n)!
    - It does not change the complexity class!
- Suppose there exists a machine that can execute ANY number of functions in parallel
  - Assume that a single function call takes polynomial (P) time
  - Overall running time?
    - Still polynomial time!

This is basically kind of Quantum Computing!

What is a problem in NP?

- A problem that can be solved by such a non-deterministic machine in polynomial time
- Examples:
  - Many, many, many …



Major property of NP problems

- It is easy (=P time) to **verify** whether an assignment is a solution
- It is significantly harder to **find** the solution

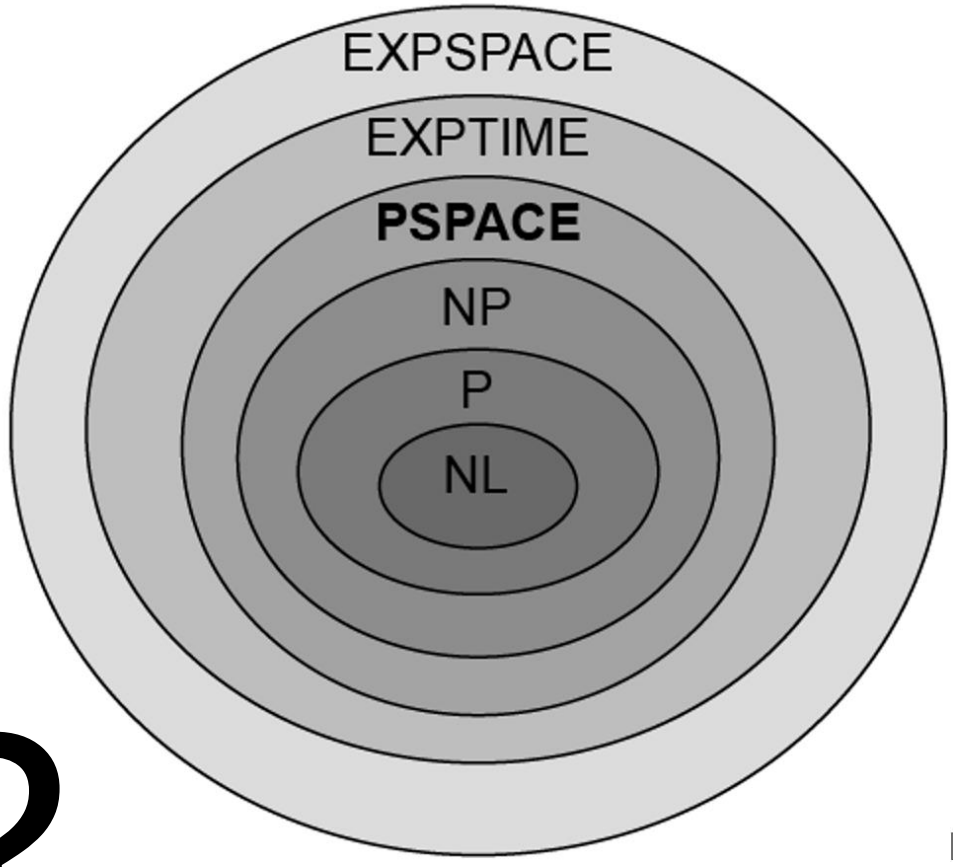
NP-hard problem: Example

- Boolean formulae, consisting of:
  - Variables
  - Negation
  - Conjunction
  - Disjunction
- Question: Is a given formula satisfiable?
  - Hard for simplified instances/normal forms
- Brute force solution?

NP-hard problem: Example

- Boolean formulae, consisting of:
  - Variables
  - Negation
  - Conjunction
  - Disjunction
- Question: Is a given formula satisfiable?
  - Hard for simplified instances/normal forms
- Brute force solution?
- There exist heuristics /approximations which lead to solutions of problems with 100000 variables on nowadays computers

The biggest (open) question in Computer Science is ...



P=NP?

Why is this question so interesting?

- There is a WHOLE load of problems which are known to be in NP. Examples? Here you go …
- Graphs:
  - Capacitated Minimum Spanning Tree
  - Degree-constrained Spanning Tree
  - Graph coloring
  - Dominating Set
  - Clique Cover
  - Graph partitioning
  - Vertex cover
  - Etc.

Why is this question so interesting?

- There is a WHOLE load of problems which are known to be in NP. Examples? Here you go …
- Mathematical Programming:
  - Knapsack
  - Subset Sum
  - 3-partition
  - Bin-packing
  - Traveling salesman problem
  - Integer programming
  - Non-convex quadratic programming
  - etc.

Why is this question so interesting?

- There is a WHOLE load of problems which are known to be in NP. Examples? Here you go …
- Others:
  - Boolean satisfiability
  - Circuit satisfiability
  - Cyclic ordering
  - Set packing
  - **Many facility location problems**
  - **Many routing problems**
  - **Many scheduling problems**
  - etc.

Why is this question even more interesting?

- People have tried for 40-50 years to prove P=NP or P!=NP
  - Nobody succeeded so far
  - Many smart people spent half of their lives on this question!
- Benefit:
  - These attempts induced many new concepts in computer science

Does this problem need more advertisement?

Why is this question REALLY interesting?

- If you solve **any** of the NP problems in P time, you can solve **all** others in P time!

Why is this question REALLY interesting?

- If you solve **any** of the NP problems in P time, you can solve **all** others in P time!

Problem Reduction

- Express one problem in terms of another problem
- A simple analogy:
  - We do not know how to multiply, but we can compute the square of numbers
  - Can we still solve multiplication?

Problem Reduction

- Express one problem in terms of another problem
- A simple analogy:
  - We do not know how to multiply, but we can compute the square of numbers
  - Can we still solve multiplication?

$$a \times b = \frac{\left((a+b)^2 - a^2 - b^2\right)}{2}$$

- We reduce multiplication to squaring …

Why is this question REALLY interesting?

- For all problems in NP, it has been shown that there exists a P time transformation procedure to at least one other NP problem
  - If you solve one problem in P time, you can transfer the results to any other problem in P time!
  - **Rationale: Many problems involve similar structures!**
    - Selecting nodes/links
    - Selecting numbers
    - Selecting variables

One example transformation

First problem

- *3-SATISFIABILITY (3SAT)*
  Instance: Set U of variables, a collection C of clauses over U such that each clause c in C has size exactly 3.
  Question: Is there a truth assignment for U satisfying C?

$C =$ $(\neg v_1$ OR v_2 OR $\neg v_3)$ AND $(v_1$ OR $\neg v_2$ OR $\neg v_4)$ AND
$(\neg v_2$ OR $\neg v_4$ OR $v_5)$ AND $(v_3$ OR v_4 Or $v_5)$

First problem

- *3-SATISFIABILITY (3SAT)*
  Instance: Set U of variables, a collection C of clauses over U such that each clause c in C has size exactly 3.
  Question: Is there a truth assignment for U satisfying C?

$C =$ $(\neg v_1$ OR v_2 OR $\neg v_3)$ AND $(v_1$ OR $\neg v_2$ OR $\neg v_4)$ AND
$(\neg v_2$ OR $\neg v_4$ OR $v_5)$ AND $(v_3$ OR v_4 Or $v_5)$

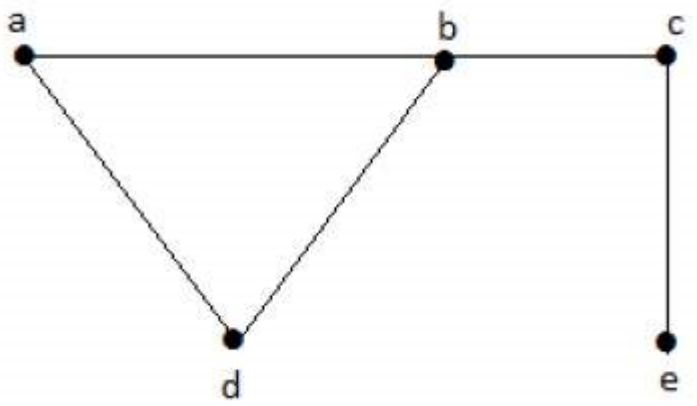
$C = \{\neg v_1, v_2, \neg v_3\}, \{v_1, \neg v_2, \neg v_4\},$
$\{\neg v_2, \neg v_4, v_5\}, \{v_3, v_4, v_5\}$

Let us assume that somebody smart proved that 3SAT is NP-hard!

Second problem

- *The Independent Set Problem*

Problem: Given a graph $G = (V, E)$ and an integer k, is there a subset S of at least k vertices such that no $e \in E$ connects two vertices that are both in S ?



Second problem

- *The Independent Set Problem*

Problem: Given a graph $G = (V, E)$ and an integer k, is there a subset S of at least k vertices such that no $e \in E$ connects two vertices that are both in S ?

Theorem: Independent Set is NP-complete.

Proof: How can we prove that it is a NP hard problem?

Second problem

- *The Independent Set Problem*

Problem: Given a graph $G = (V, E)$ and an integer k, is there a subset S of at least k vertices such that no $e \in E$ connects two vertices that are both in S ?

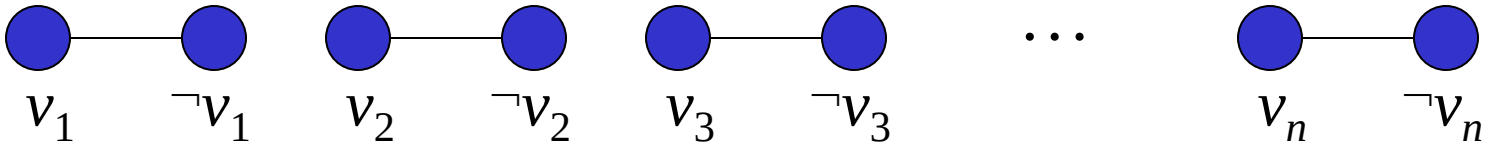
Theorem: Independent Set is NP-complete.

Proof: How can we prove that it is a NP hard problem?

=> Solve 3SAT using Independent Set Problem

Reducing 3-SAT to Independent Set

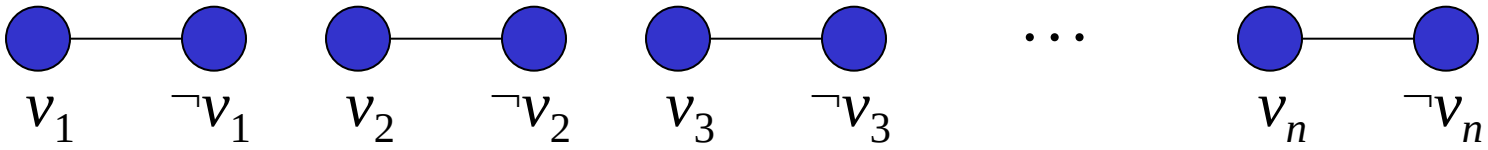
For each variable, we can create two vertices:



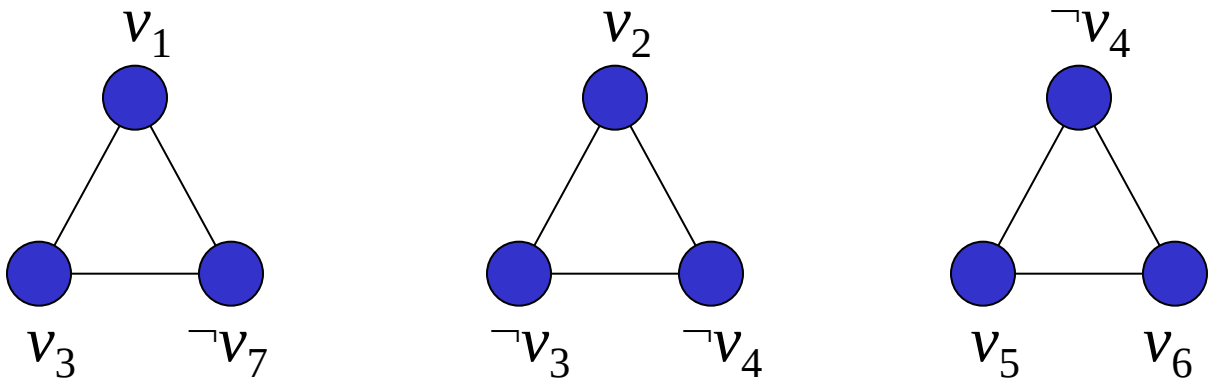
If we connect a variable and its negation, we can be sure that only *one* of them is in the set. In all, we must have n vertices in S to be sure all variables are assigned.

This will handle the binary true-false values; how can we also make sure that all of the clauses are fulfilled?

Including *Clauses* in the Reduction



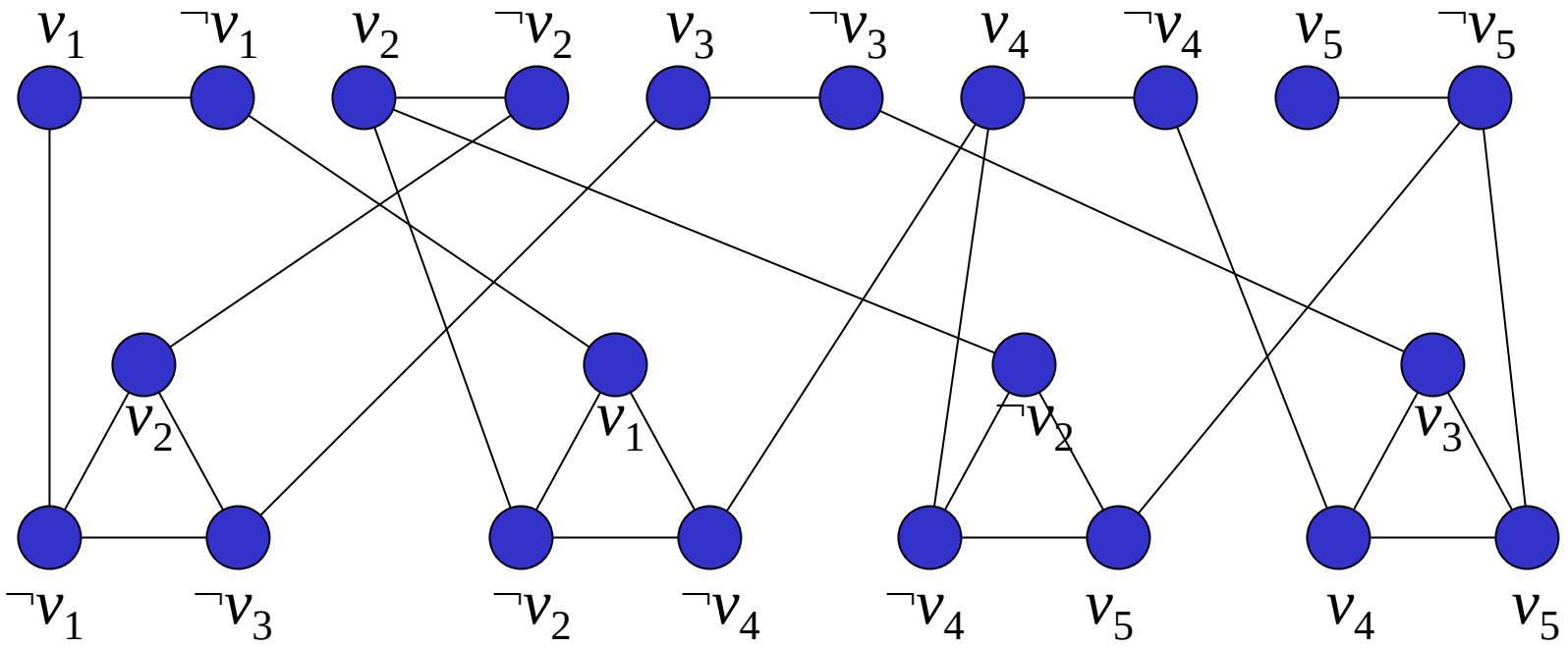
We can consider the clauses as triangles:



Each clause has at least one true value. On the other hand, at most one vertex in a triangle can be in the independent set. So how do we tie these together?

Tying it all together...

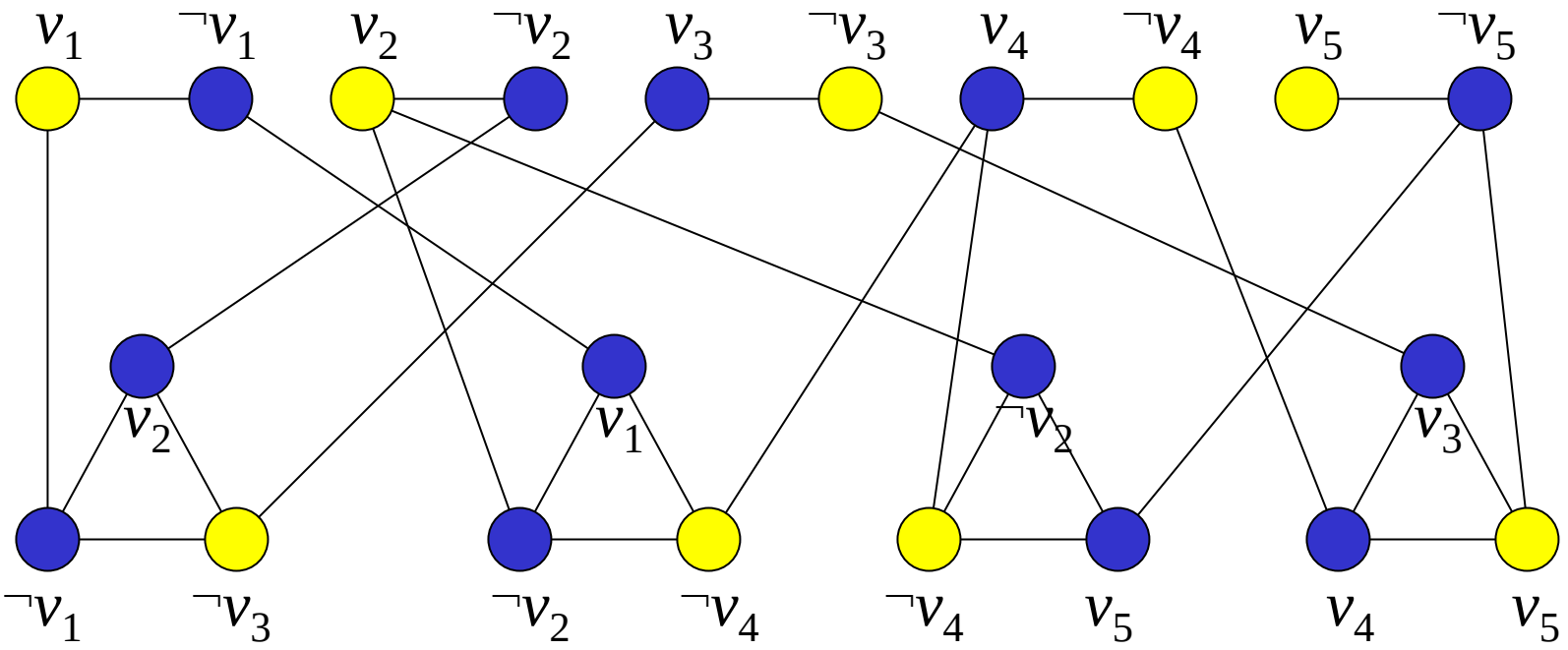
$$C = \{\neg v_1, v_2, \neg v_3\} \ , \ \{v_1, \neg v_2, \neg v_4\} \ ,$$
$$\{\neg v_2, \neg v_4, v_5\} \ , \ \{v_3, v_4, v_5\}$$



If there are n variables and m clauses, then finding the satisfiability is the same as finding n+m independent vertices.

$$C = \{\neg v_1, v_2, \neg v_3\} \ , \ \{v_1, \neg v_2, \neg v_4\} \ , \ \{\neg v_2, \neg v_4, v_5\} \ , \ \{v_3, v_4, v_5\}$$

Here you go:



Why is this question REALLY interesting?

- Wouldn't it be nice to solve all computer science problems at once, by solving a single problem only?
- Nowadays, people believe that P!=NP
  - No proof yet
  - If you were able to prove P!=NP, you would get (at least) one million USD, and much more fame!

Why is this question REALLY interesting?

- Wouldn't it be nice to solve all computer science problems at once, by solving a single problem only?
- Nowadays, people believe that P!=NP
  - No proof yet
  - If you were able to prove P!=NP, you would get (at least) one million USD, and much more fame!

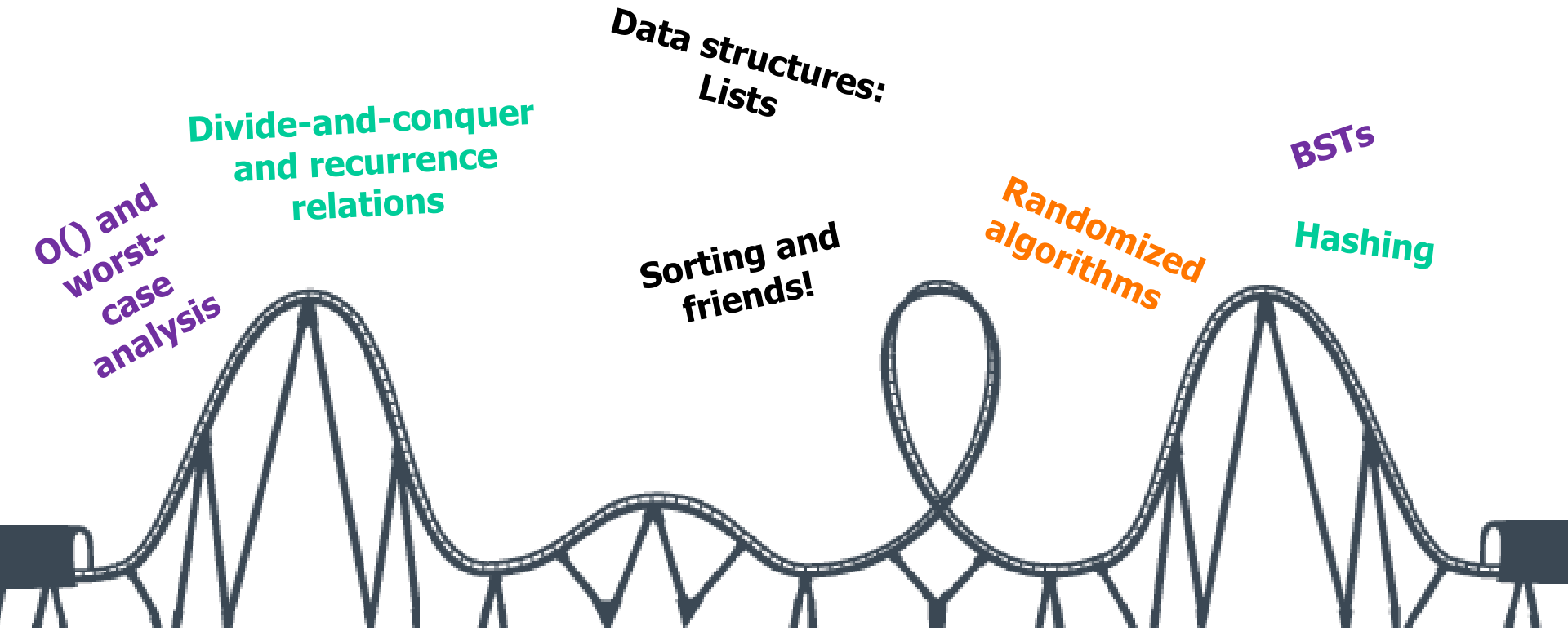
Question: Is there an optimal, fast algorithm for my problem?

Answer: If your problem is NP-hard, then very likely not!

(Sorry, that the very last message of this lecture is so … "inexact". ☹)

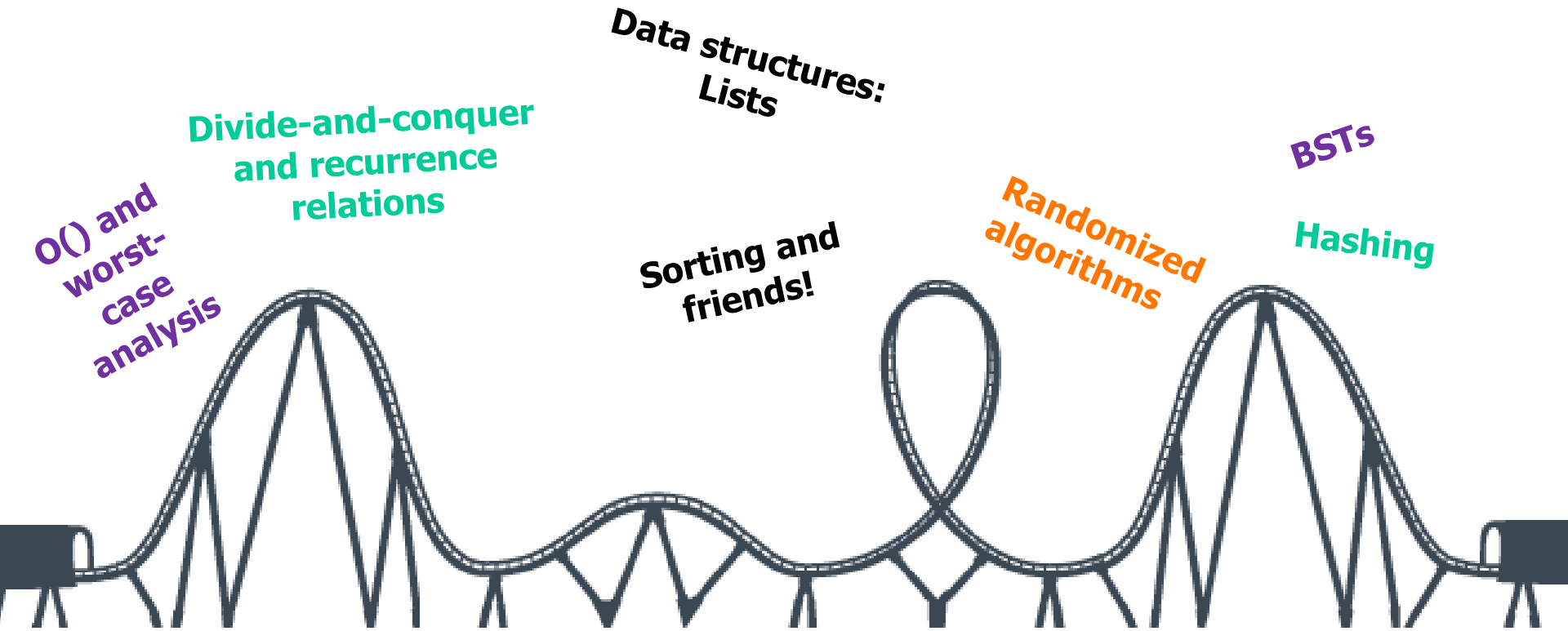
Summary

What happened?



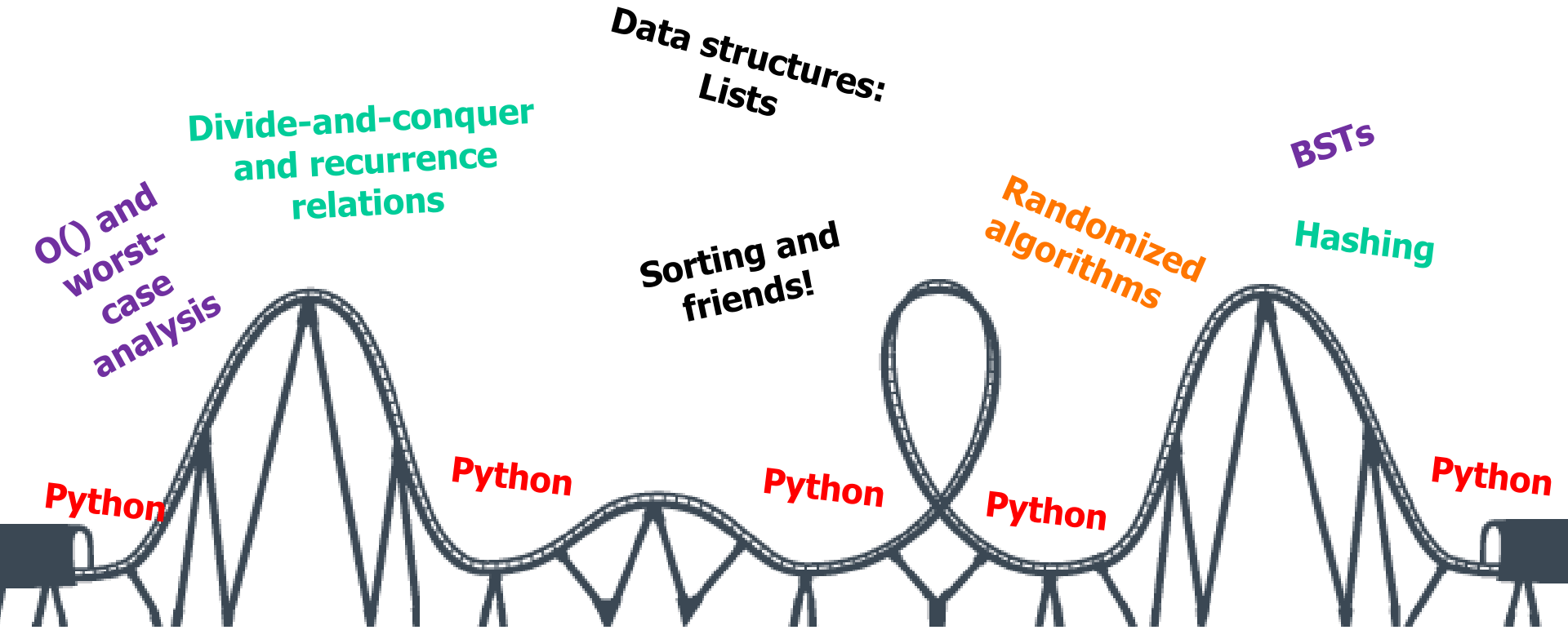
What happened?

Anything missing?



What happened?

Anything missing?



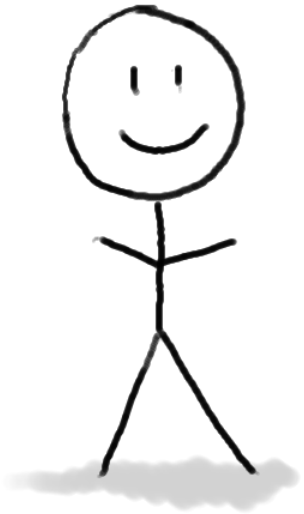
You have seen (and hopefully learned) a lot

- Let's have a quick tour of the core concepts
  - ICSP in 10 slides, instead of ~1300 slides

Algorithm design and analysis

Main theme of the lecture:

Can I do better?



Algorithm designer

We needed more details!

$$T(n) = O(f(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ s.t. \ \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot f(n)$$

Does it work?
Is it fast?

What does that mean??

Worst-case analysis

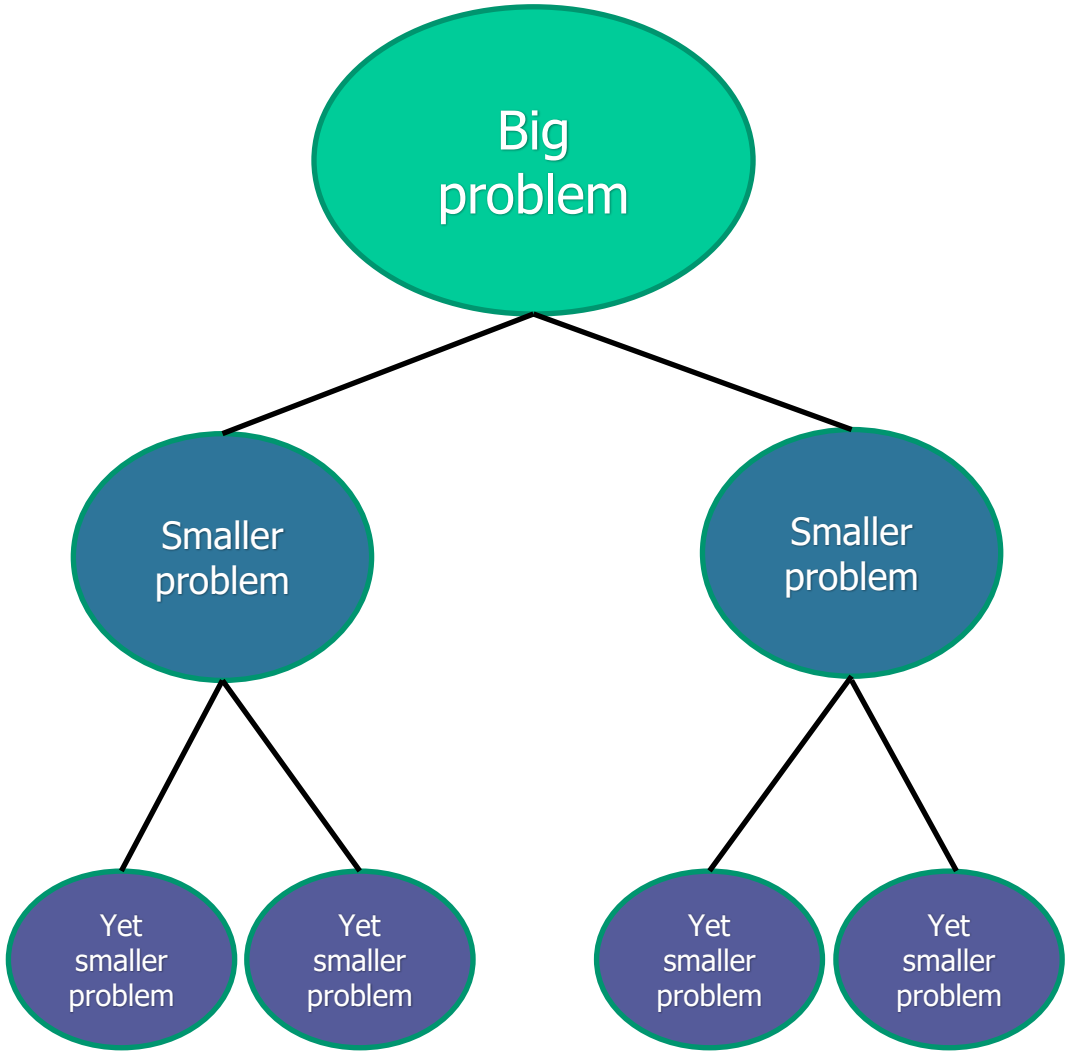
Here is an input!

big-O notation

$$T(n) = O(f(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot f(n)$$

Design paradigm: Divide and Coquer

- Like MergeSort!
- Or Karatsuba's algorithm!
- Or k-Select!
- How do we analyze these?

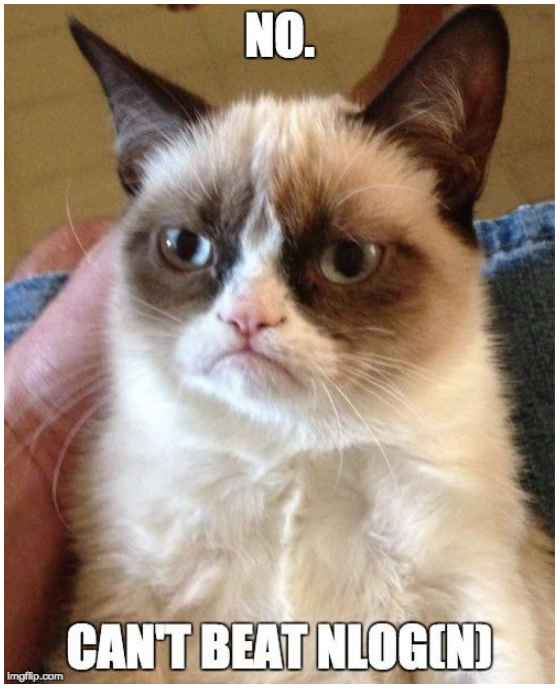


By careful analysis!

Useful shortcut, the **master method** is.

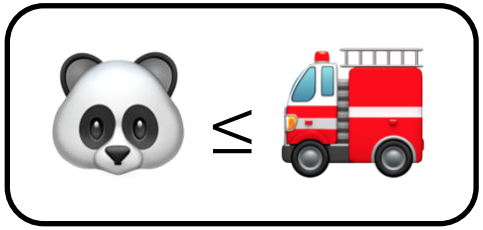
All this sorting makes you wonder …? Can we do better?

- Depends on who you ask:



- **CountingSort** takes time O(n) if the objects are, for example, small integers!

- Can't do better in a **comparison-based** model.



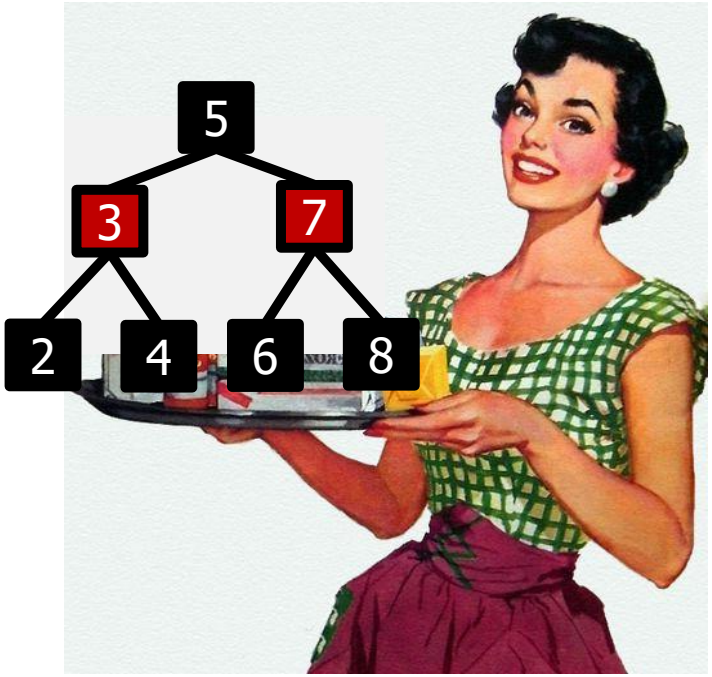
Binary Search Trees

- Useful data structure!
- Especially the self-balancing ones!

Red-Black tree!

Maintain balance by stipulating that **black nodes** are balanced, and that there *aren't too many* **red nodes**.

It's just good sense!

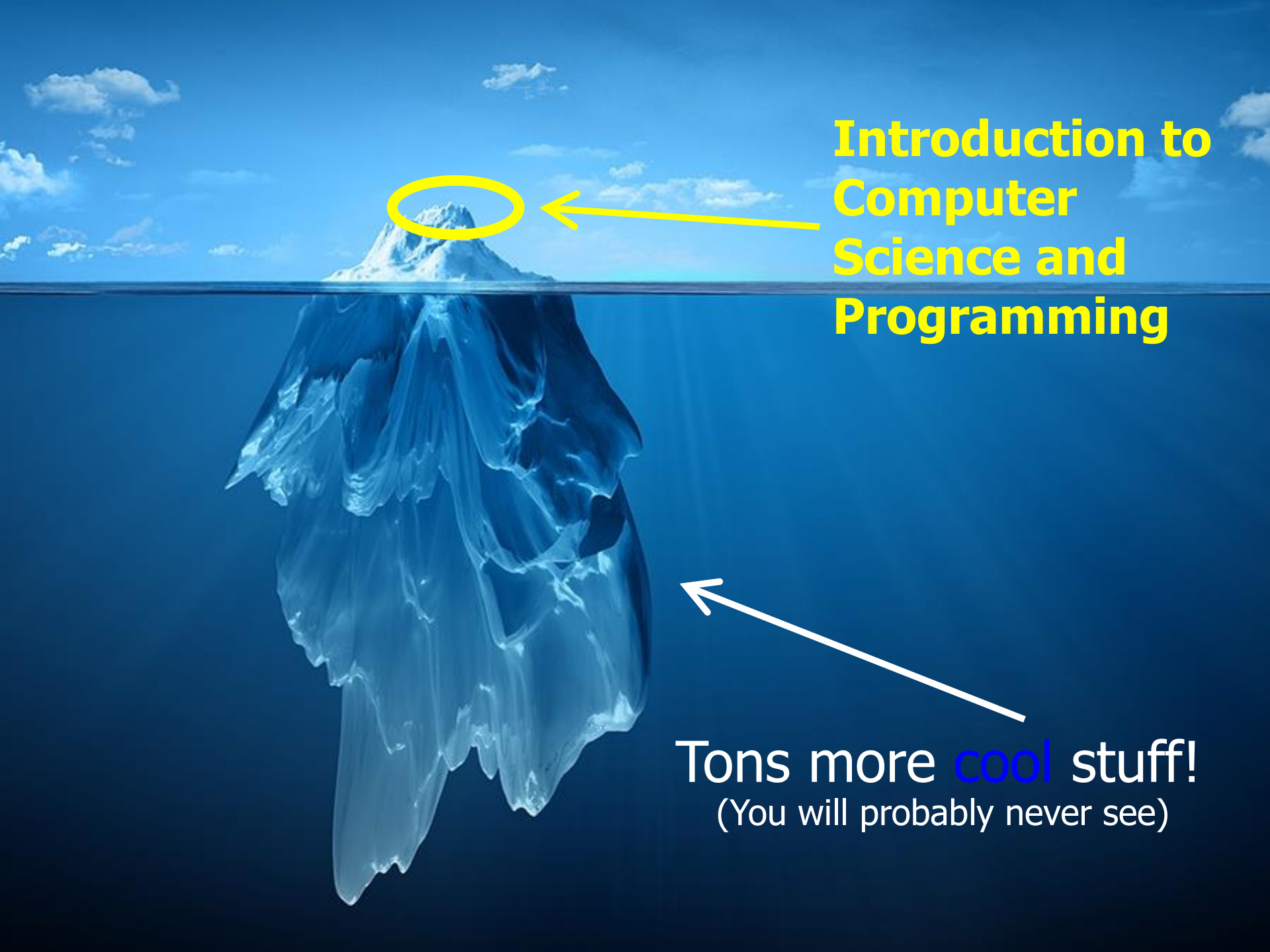


Hash tables: Another way to store objects

The universe

hash function h

Some buckets


Introduction to Computer Science and Programming
Tons more cool stuff!
(You will probably never see)

Exam

Types of tasks (not exhaustive):

- Theory:
  - Proof that XYZ
  - Formally define the notion of XYZ
- Programming (Python)
  - **Design an algorithm for a given problem XYZ and implement it**
  - What is the time complexity of a specific algorithm/program?
  - Finding errors in a program
- Applied tasks
  - Apply algorithm XYZ to the following input and show all intermediate steps

In total: 120 minutes

Material

- You are not allowed to use any material, including
  - Text books
  - Own paper, summaries
  - Mobile phones, tablets, calculator, etc.
  - Solutions of neighbors
- You are allowed to use:
  - Pencil / Pen
  - Tissue (in reasonable amounts ☺)

Questions?

Any feedback from your side?

- Did you get along well with the book?
- ~~How do you like the recap?~~
- What to improve?
- Which topic was too easy?
- Which topic was too difficult?
- Any improvements regarding the lab classes?

Thank you very much!