Algorithms & Complexity
Lecture 2: Sorting

September 24, 2019

CentraleSupélec / ESSEC Business School

Dimo Brockhoff

Inria Saclay – Ile-de-France

Correction from Last Lecture

! The definition of the O-notation had a mistake related to where the absolute value was !

- it reads correctly $|f(n)| \leq c \cdot g(n)$ instead of $f(n) \leq c \cdot |g(n)|$ in the definition [corrected in old slides on the web]
- it definitely makes more sense like that:
  - $-n = O(n)$ i.e. $-n$ increases at most as quickly as n

Course Overview

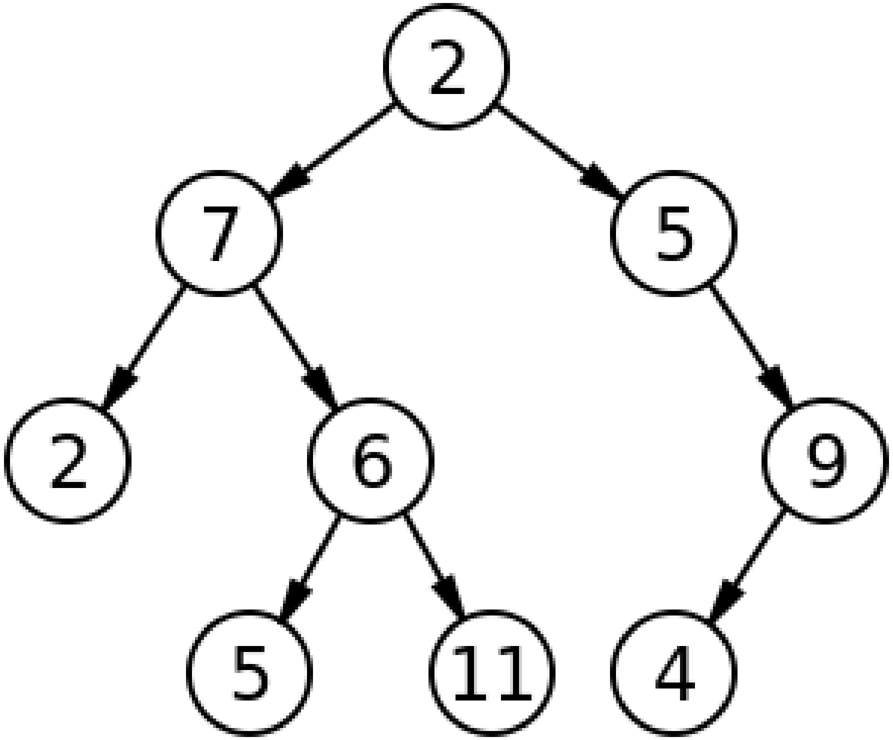
	Thu		Topic
	Thu, 12.09.2019	PM	Introduction, Combinatorics, O-notation, data structures
➨	Tue, 24.09.2019	PM	Sorting algorithms I
	Tue, 1.10.2019	PM	Sorting algorithms II, recursive algorithms
	Tue, 8.10.2019	PM	Greedy algorithms
	Tue, 15.10.2019	PM	Dynamic programming
	Thu, 31.10.2019	AM	Randomized Algorithms and Blackbox Optimization
	Tue, 5.11.2019	PM	Complexity theory I
	Tue, 26.11.2019	PM	Complexity theory II
	Tue, 17.12.2019	AM	Exam (written)

Quick Recap

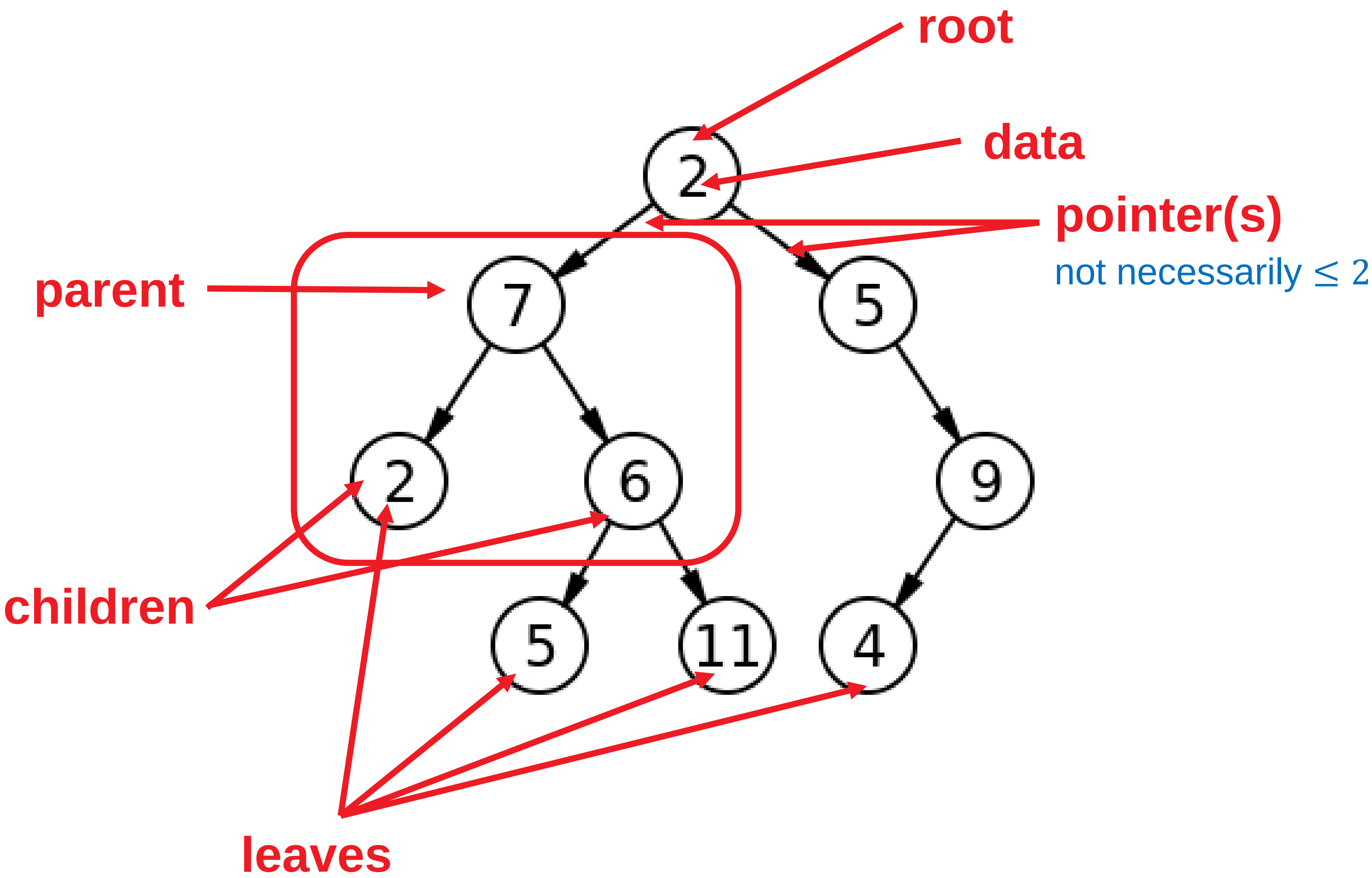
- Basics of combinatorics and the O-notation
- Data structures
  - Arrays: fast access, slow search, no insert
  - Lists: slow access, slow search, but insert/remove in constant time
    - Hence python lists are implemented as dynamic arrays (once array is full, a larger chunk of memory gets allocated) http://www.laurentluce.com/posts/python-list-implementation/
  - Trees: log(n) access, log(n) add/remove [today]
  - Dictionaries: we will see ☺

Trees

Trees



Trees


root
data
pointer(s)
not necessarily ≤ 2
parent
children
leaves
2
7
5
2
6
9
5
11
4

Trees are Special Graphs

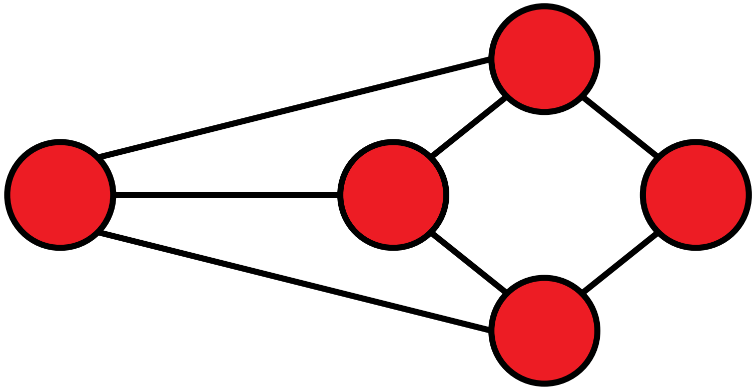
For a more formal definition, we need to introduce the concept of graphs…

Basic Concepts of Graph Theory

[following for example http://math.tut.fi/~ruohonen/GT_English.pdf]

Graphs

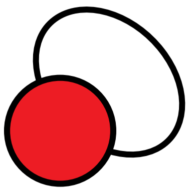
Definition 1 *An undirected graph G is a tupel $G = (V, E)$ of edges $e = \{u, v\} \in E$ over the vertex set V (i.e., $u, v \in V$).*



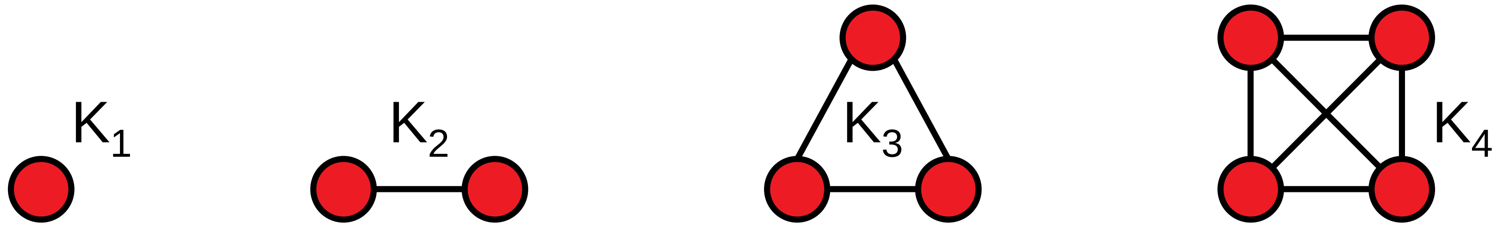
- vertices = nodes
- edges = lines
- Note: edges cover two *unordered* vertices (*undirected* graph)
  - if they are *ordered*, we call G a *directed* graph with edges $e = (u, v)$

Graphs: Basic Definitions

- G is called *empty* if E empty
- u and v are *end vertices* of an edge {u,v}
- Edges are *adjacent* if they share an end vertex
- Vertices u and v are *adjacent* if {u,v} is in E
- The *degree* of a vertex is the number of times it is an end vertex
- A complete graph contains all possible edges (once):



a loop

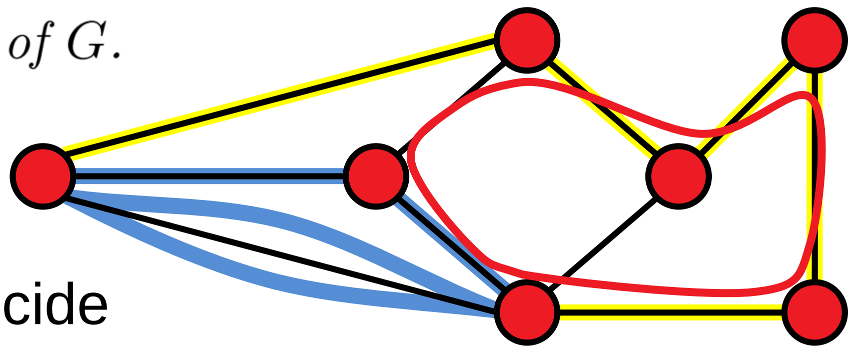


Walks, Paths, and Circuits

Definition 1 *A* walk *in a graph* $G = (V, E)$ *is a sequence*

$$v_{i_0}, e_{i_1} = (v_{i_0}, v_{i_1}), v_{i_1}, e_{i_2} = (v_{i_1}, v_{i_2}), \ldots, e_{i_k}, v_{i_k},$$

alternating vertices and adjacent edges of G*.*



A walk is

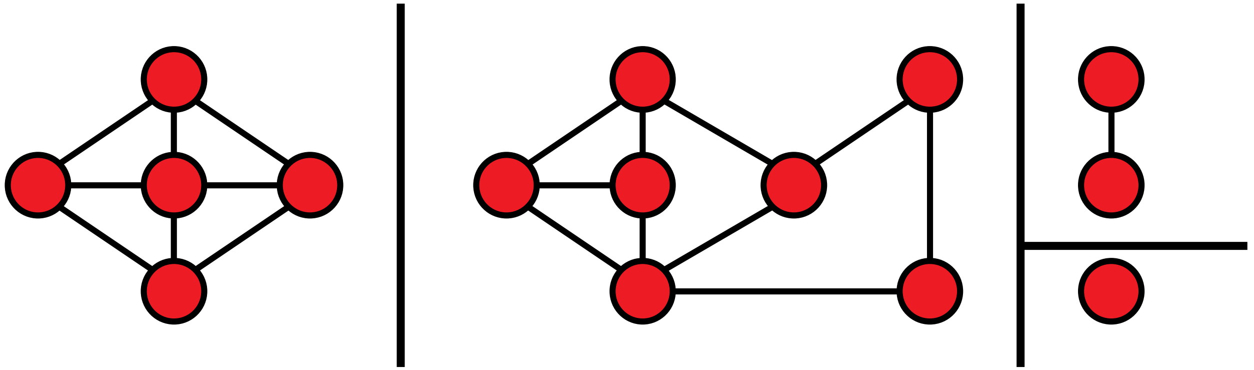
- *closed* if first and last node coincide
- a *trail* if each edge traversed at most once
- a *path* if each vertex is visited at most once

- a closed path is a *circuit* or *cycle*
- a closed path involving all vertices of G is a *Hamiltonian cycle*

Graphs: Connectedness

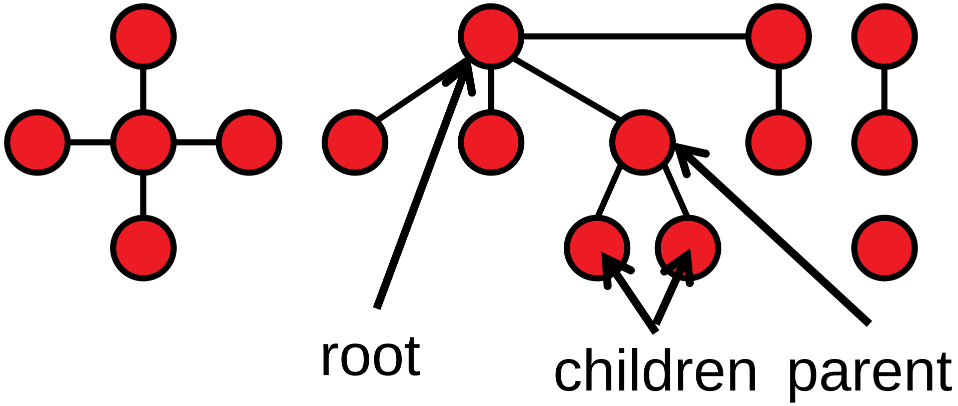
- Two vertices are called *connected* if there is a walk between them in G
- If all vertex pairs in G are connected, G is called connected

- The *connected components* of G are the (maximal) subgraphs which are connected.

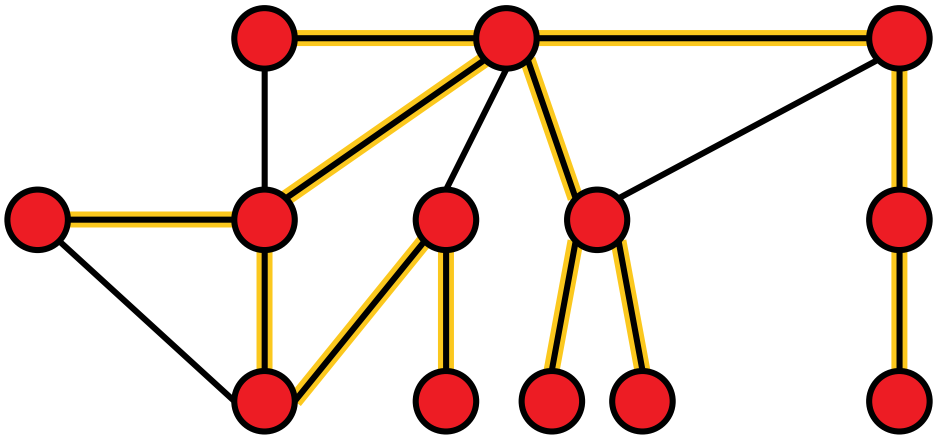


Trees and Forests

- A *forest* is a cycle-free graph
- A *tree* is a connected forest



A *spanning tree* of a connected graph G is a tree in G which contains all vertices of G



Depth-First Search (DFS)

Sometimes, we need to traverse a graph, e.g. to find certain vertices

Depth-first search and breadth-first search are two algorithms to do so

Depth-first Search (for undirected/acyclic and connected graphs)

❶ start at any node x; set i=0

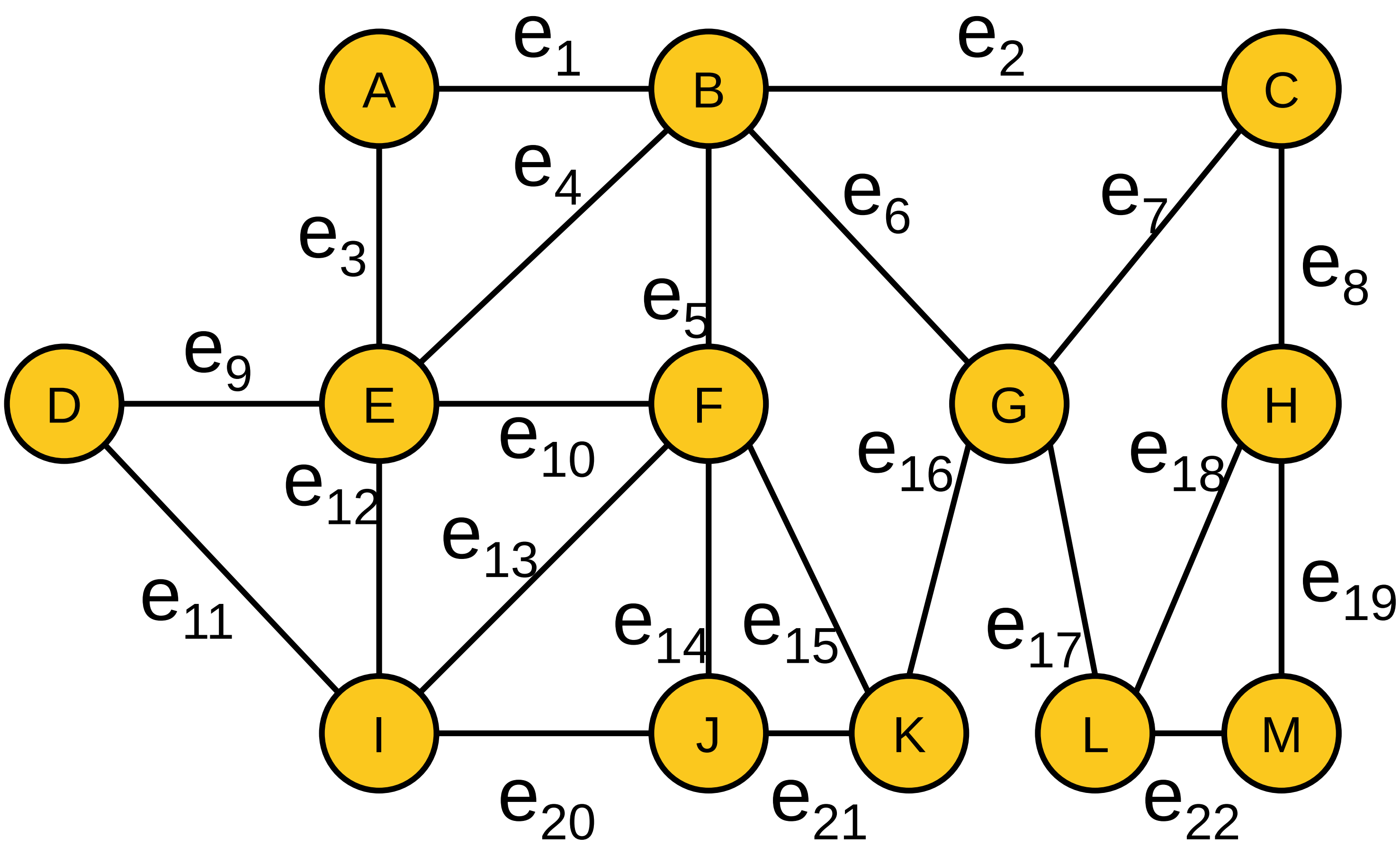
❷ as long as there are unvisited edges {x,y}:

- choose the next unvisited edge {x,y} to a vertex y and mark x as the parent of y
- if y has not been visited so far: i=i+1, label y as the node visited at iteration i, and continue the search at x=y in step 2
- else continue with next unvisited edge of x

❸ if all edges {x,y} are visited, we continue with x=parent(x) at step 2 or stop if x equals the starting node v0

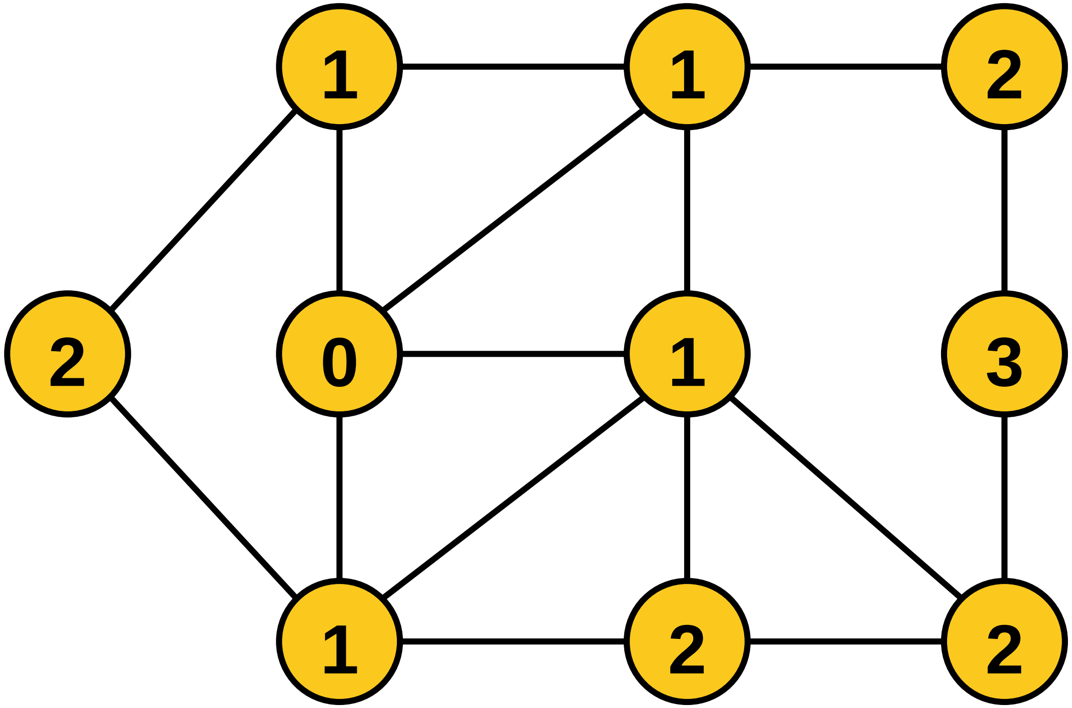
DFS: Stage Exercise

Exercise the DFS algorithm on the following graph!



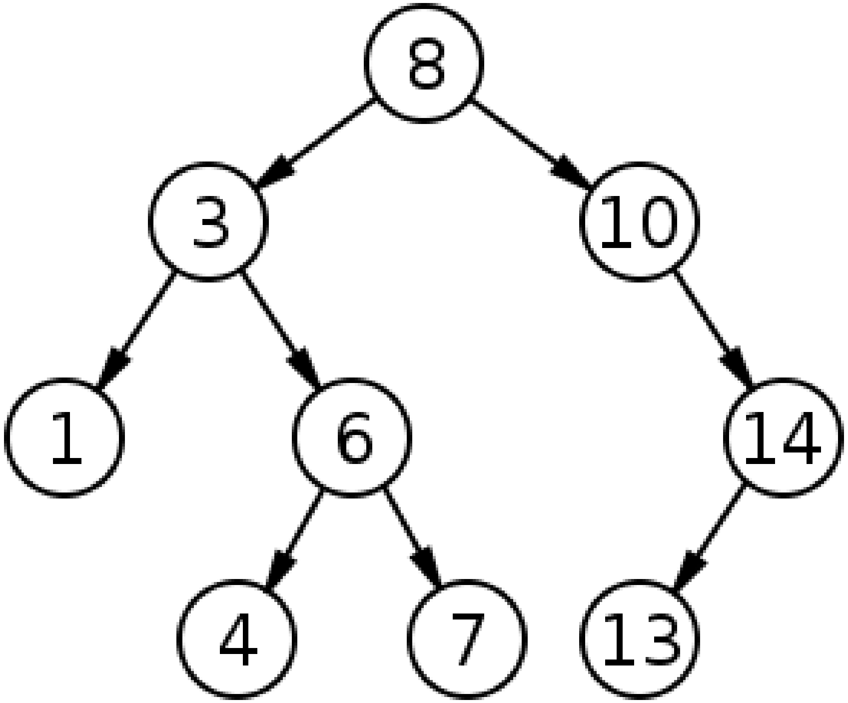
Breadth-first Search (for undirected/acyclic and connected graphs)

1. start at any node x, set i=0, and label x with value i
2. as long as there are unvisited edges {x,y} which are adjacent to a vertex x that is labeled with value i:
   - label all vertices y with value i+1
3. set i=i+1 and go to step 2



Binary Search Tree

- a tree with degree ≤ 2
- children sorted such that the left subtree always contains values smaller than the corresponding root and the right subtree only values larger



Class Exercise: Filling a Binary Search Tree

Round 1:

give an integer to be filled into our tree

Round 2:

tell where the next integer inserts

Binary Search Tree: Complexities

Search

- similar to binary search in array (go left or right until found)
- $O(\log(n))$ if tree is well balanced
- $\Theta(n)$ in worst case (linear list)

Insertion

- first like search to determine the parent of the new node
- then add in $O(1)$ [we are always at a leaf or have an “empty child”]

Remove (more tricky)

- if node has no child, remove it
- if node has a single child, replace node by its child
- if node has two children: find left-most tree entry L larger than the to-be-removed node, copy its value to the to-be-removed node, and remove L according to the two above rules
- cost: $O(\text{tree depth})$, in worst case: $\Theta(\mathrm{n})$

Binary Trees: Can We Do Better?

Binary Search Tree

average case (random inserts)			worst case		
search	insert	delete	search	insert	delete
$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Guarantee a balanced tree:

- AVL trees
- B trees
- Red-Black trees
- …

average case (random inserts)			worst case		
search	insert	delete	search	insert	delete
$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

Can We Do Even Better on Average?

Balanced Trees

average case (random inserts)			worst case		
search	insert	delete	search	insert	delete
$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

?

average case (random inserts)			worst case		
search	insert	delete	search	insert	delete
$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Dictionaries

In python:

```
my_dict = {'Joe': 113, 'Pete': 7, 'Alan': '110'}
print("my_dict['Joe']: " + my_dict['Joe'])
```

gives `my_dict['Joe']: 113` as output

- the immutables **`'Joe'`**, **`'Pete'`**, and **`'Alan'`** are the keys
- **113**, **7**, and **110** are the values (or the stored data)

Next: Why dictionaries and how are they implemented?

Dictionaries

Where is Alan?

Where is Alan?

- Go through all offices one by one?

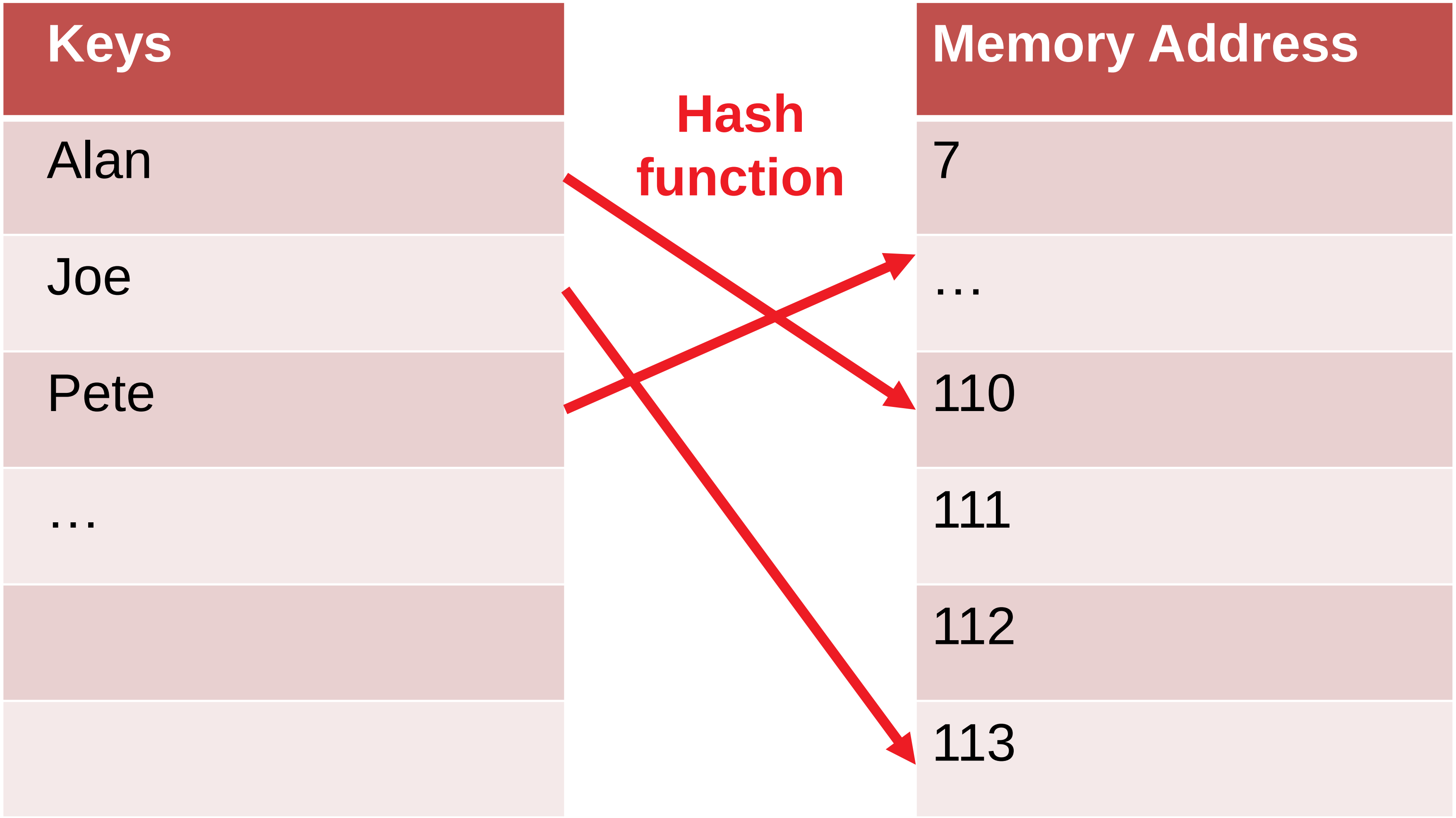
like in list and array

- No, you would ask the receptionist for the office number

Dictionaries Implemented as Hashtables

Names
Alan
Joe
Pete
…

Offices
7
…
110
111
112
113



Possible hash function: $h = z \textbf{ mod } n$

Hash Functions

…should be

- deterministic: find data again
- uniform: use allocated memory space well
  [more tricky with variable length keys such as strings]

Problems to address in practice:

- how to deal with collisions (e.g. via multiple hash functions)
- deleting needs to insert dummy keys when a collision appeared
- what if the hash table is full? → resizing

All this gives a constant average performance in practice
and a worst case of $\Theta(n)$ for insert/remove/search

Not more details here, but if you are interested:

For more details on python's dictionary:
https://www.youtube.com/watch?v=C4Kc8xzcA68

What Have We Learned?

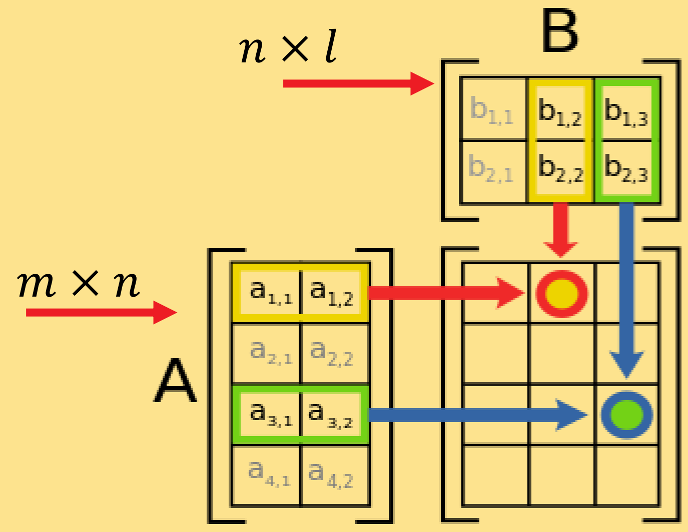
- Combinatorics: basic ways of counting things
- O-notation: how to formalize classes of asymptotic function growth
- Basic data structures and their operations
  - arrays
  - lists
  - (binary search) trees
  - dictionaries / hash tables

see also https://www.bigocheatsheet.com/

- And along the way: graph theory, DFS, and BFS

discussion home exercises

Exercise 1: Matrix Multiplication



- $c_{ij} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$
- naïve implementation:

  for $i = 1$ **to** m **do**:
    for $j = 1$ **to** l **do**:
      $c_{ij} = 0$
      for $k = 1$ **to** n **do**:
        $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$

- computation per cell: n additions and n multiplications
- has to be done for all $m \times l$ cells
- in total: $m \cdot l \cdot n$ additions and $m \cdot l \cdot n$ multiplications
- $\Theta(n^3)$ if $k = l = n$
- interesting: we can do better:
  $O\left(n^{\log 7}\right) = O(n^{2.807\ldots})$ by Strassen (1968)
  even $O(n^{2.3728639})$ by Le Gall (2014)

Discussion Home Exercise

Exercise 2: Tennis Event


Mad melone

- 2 players: trivial
- 4 players:
  - first round: 2 games
  - final (winners from first games) gives best player
  - another game needed (!): winner of the two losers against best is 2nd best
  - 4 games in total
- with $n = 2^k$ players: k rounds kicks out half of the players with $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 4 + 2 + 1 = n - 1$ games to find out best
- Then $k - 2 = O(\log(n))$ more games needed to find second best as best among the losers against overall best

Discussion Home Exercise

Exercise 3: Tennis Event II

No change in asymptotic number of $\Theta(n)$ games, because already finding out about the best player needs $\Theta(n)$ games

Exercise 4: O-Notation

$$O(f_1) + O(f_2) = O(f_1 + f_2)$$

Proof:

- choose $g_1 \in O(f_1)$ and $g_2 \in O(f_2)$ arbitrarily
- i.e. we have constants $n_1, n_2, c_1, c_2 > 0$ such that
  $g_1(n) \leq c_1 \cdot f_1(n)$ for all $n > n_1$ and
  $g_2(n) \leq c_2 \cdot f_2(n)$ for all $n > n_2$
- but with $c_+ = \max\{c_1, c_2\}$ then also

$$|g_1(n) + g_2(n)| \leq |g_1(n)| + |g_2(n)|$$
$$\leq c_1 \cdot f_1(n) + c_2 \cdot f_2(n) \leq c_+ \cdot (f_1(n) + f_2(n))$$

Discussion Home Exercise

Exercise 4: $\boldsymbol{O}$-Notation

$$O(f_1) - O(f_2) \neq O(f_1 - f_2)$$

Proof by counter example:

- use $f_1(n) = f_2(n) = n$
- let $g_1(n) = n$ and $g_2(n) = 0$
- now we have that $g_1 \in O(f_1)$ and $g_2 \in O(f_2)$
- but $g_1 + g_2 = n + 0 \notin O(f_1 + f_2) = O(0)$


now: sorting...
BY NC ND
jwhittenburg

Exercise: Sorting

Aim: Sort a set of numbers

Questions:

- What is the underlying algorithm you used?
- How long did it take to sort?
  - What is a good measure?
- Is there a better algorithm or did you find the optimal one?

Overview of Today's Lecture

Sorting

- Insertion sort
- Insertion sort with binary search
- Mergesort
- Timsort idea
- Quicksort idea

Exercise

- Comparison of sorting algorithms

Essential vs. Non-Essential Operations

In sorting, we distinguish

- comparison- and non-comparison-based sorting
- in the former, we distinguish further:
  - comparisons as essential operations
    - they are comparable over computer architectures, operating systems, implementations, (historic) time
    - they can take more time than other operations, e.g. when we compare trees w.r.t. their lexicographic DFS sorting
  - other non-essential operations: additions, multiplications, shifts/swaps in arrays, …

Insertion Sort

Idea:

for k from 1 to n-1:

- assume array a[1]…a[k] to be sorted
- insert a[k+1] correctly into a[1]…a[k+1]

6 5 3 1 8 7 2 4

see also https://en.wikipedia.org/wiki/Insertion_sort

Insertion Sort: Analysis

Worst case:

- reverse ordering: insert always to the beginning
- then $1 + 2 + 3 + \cdots + (n-1) \ = \Theta(n^2)$ comparisons needed

Average Case:

- even here: $\Theta(n^2)$ comparisons needed (without proof)

Insertion Sort with Binary Search

Idea for an improved version:

use binary search for the right position of new entry in sorted subarray

- to insert array element $a[i]$, we need $\lceil \log(i+1) \rceil$ comparisons in worst case (= depth of the binary tree search)
- overall, therefore

$$\sum_{1 \le i \le n-1} \lceil \log(i+1) \rceil = \sum_{2 \le i \le n} \lceil \log(i) \rceil < \log(n!) + n$$

comparisons are needed

- from last time, we know that

$$\log(n!) \le e n^{n+\frac{1}{2}} e^{-n} = n\log(n) - n\log(e) + O(\log(n))$$

in total, insertion sort with binary search needs

$$n\log(n) - 0.4426n + O(\log(n))$$

comparisons in the worst case.

Another Possible Sorting Idea:

- sort first and second half of the array independently
- then merge the pre-sorted halves:
    - take the smaller of the smallest two values each time

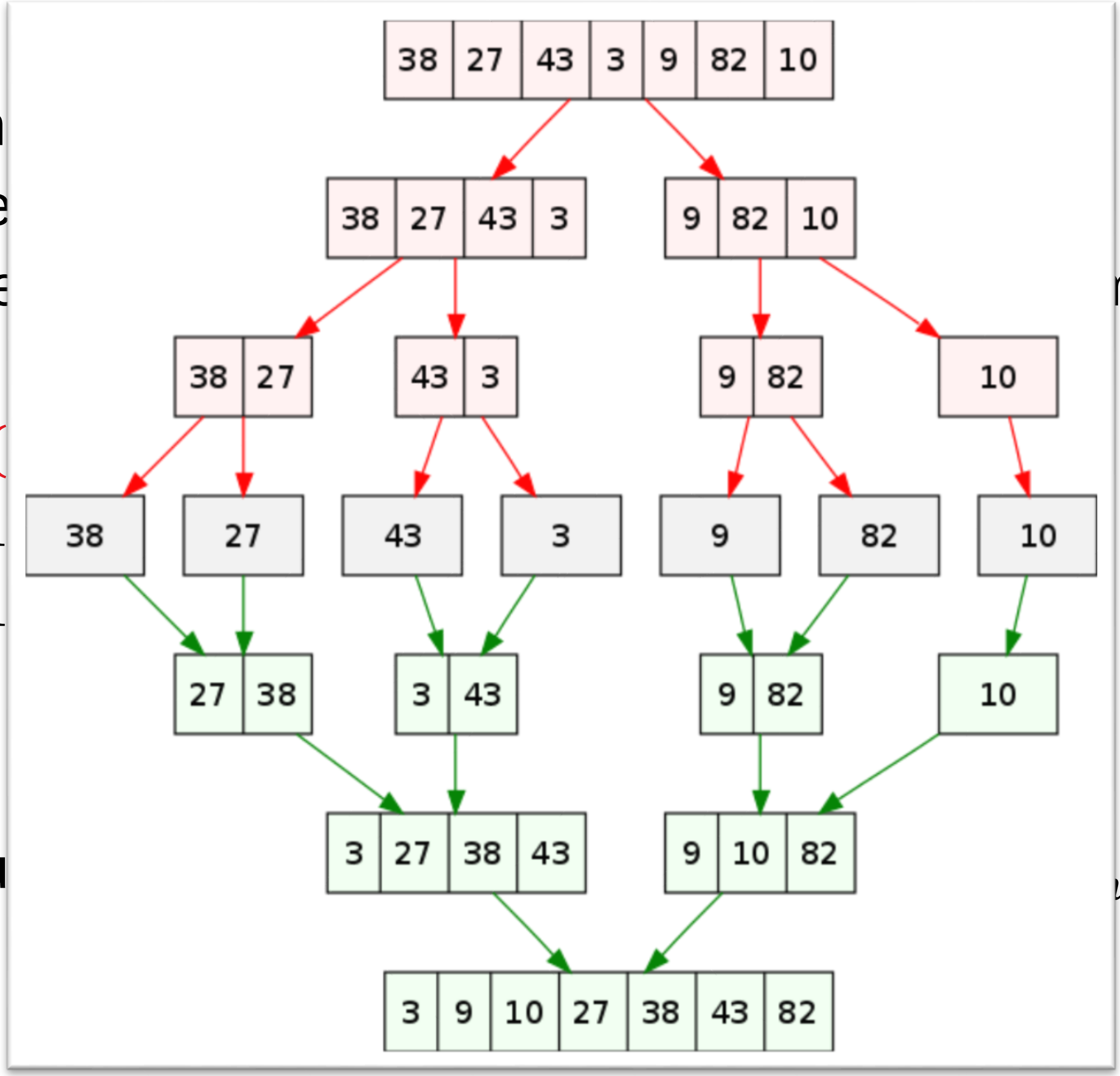
$\mathrm{Mergesort}(a_1, \ldots, a_n)$

if $n = 1$ then stop

if $n > 1$ then:

- $\left(b_1, \ldots, b_{\lceil n/2 \rceil}\right) = \mathrm{Mergesort}(a_1, \ldots, a_{\lceil n/2 \rceil})$
- $\left(c_1, \ldots, c_{\lfloor n/2 \rfloor}\right) = \mathrm{Mergesort}(a_{\lceil n/2 \rceil + 1}, \ldots, a_n)$
- return $(d_1, \ldots, d_n) = \mathrm{Merge}(b_1, \ldots, b_{\lceil n/2 \rceil}, c_1, \ldots, c_{\lfloor n/2 \rfloor})$

Another Possible Sorting Idea:



- sort first an
- then merge
  - take the … ne

Mergesort(

if $n = 1$

if $n > 1$

- $(b_1,$
- $(c_1,$
- retu … $/2\rfloor)$

Mergesort: Runtime

- the number of essential comparisons C(n) when sorting n items with Mergesort is

$$C(1) = 0, \qquad C(n) = C\left(\left\lceil \frac{n}{2} \right\rceil\right) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1$$

sorting left half | sorting right half | merging

- without proof, $C(n) = n \log(n) + n - 1$ if $n = 2^k$

Remark:

Mergesort is practical for huge data sets, that don't fit into memory

Python's Sorting: Timsort

- python uses a combination of Mergesort with insertion sort

  https://en.wikipedia.org/wiki/Timsort
- insertion sort for small arrays quicker than merging from n=1 (can be done in memory)
- in addition, Timsort searches for subarrays which are already sorted (called ”natural runs”) and that are handled as blocks
- worst case runtime of $O(n \log(n))$, best case: $O(n)$

Exercise in Python

Comparing sorting algorithms in python

Goals:

- learn about Mergesort (and how to implement it)
- observe the differences in runtime between your own Mergesort and python's internal Timsort
- learn how to do a scientific (numerical) experiment and how to report the results

Exercise in Python

TODOs:

❶ implement your own Mergesort e.g. based on lists

❷ compare the differences in runtime between your own Mergesort and python's internal Timsort (`'sorted(…)'`) on randomly generated lists of integers

❸ plot the times to sort 1,000 lists of equal length n with both algorithms for different values of $n \in \{10, 100, 1\,000, 10\,000\}$

Tip:

```
>>> import timeit
>>> timeit.timeit('your code', number=1000)
```

Another (even more important) Tip:

use the "?" to get help on a module (and "??" to inspect the code)

Conclusions

I hope it became clear...

...what is a graph, a node/vertex, an edge, ...

...what sorting is about and how fast we can do it