Collections – Part I

Nico Ludwig (@ersatzteilchen)

TOC

- Collections – Part I
  - Algorithms
    - Algorithmic Complexity
    - Analysis of Algorithms’ Costs
    - Analysis of Sorting Algorithms
    - Selection sort and Insertion sort
    - Merge sort and Quick sort
  - Divide and Conquer
  - Complexity and O-Notation

- Sources:
  - Julie Zenlisky, Stanford Course CS 106B "Programming Abstractions"
  - https://www.toptal.com/developers/sorting-algorithms
  - https://panthema.net/2013/sound-of-sorting

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Collections and Algorithms

- The topics collections and algorithms are highly coupled in programming altogether.
  - Collections and algorithms are no difficult topics by themselves, but they are difficult to give a structure to learn them.
  - This is true because:
    - Different platforms use different approaches/technologies and different terminology and categorization expressing collections and algorithms.
    - Different people of different shops have different understanding of the reality they cast into collection types. – All of these abstractions are correct!
    - But I'll try to do my best here! :) The idea is to make the audience able to ask questions like "Here in C++ I need something like Java's *Set*!".
    - The aim is to learn how certain collection types work from a 20K miles perspective, not a specific collections API!

- Collections are basically, well, collections of objects. The algorithms we'll analyze operate on collections of objects.
  - The following algorithms will use the simplest collection type: collections following the array-concept (hence, we call them "arrays").
  - Even novice programmers know how to use arrays.
  - Every language supports arrays somehow.

- In this lecture we'll discuss algorithms, which form the base for the much longer discussion of collections.

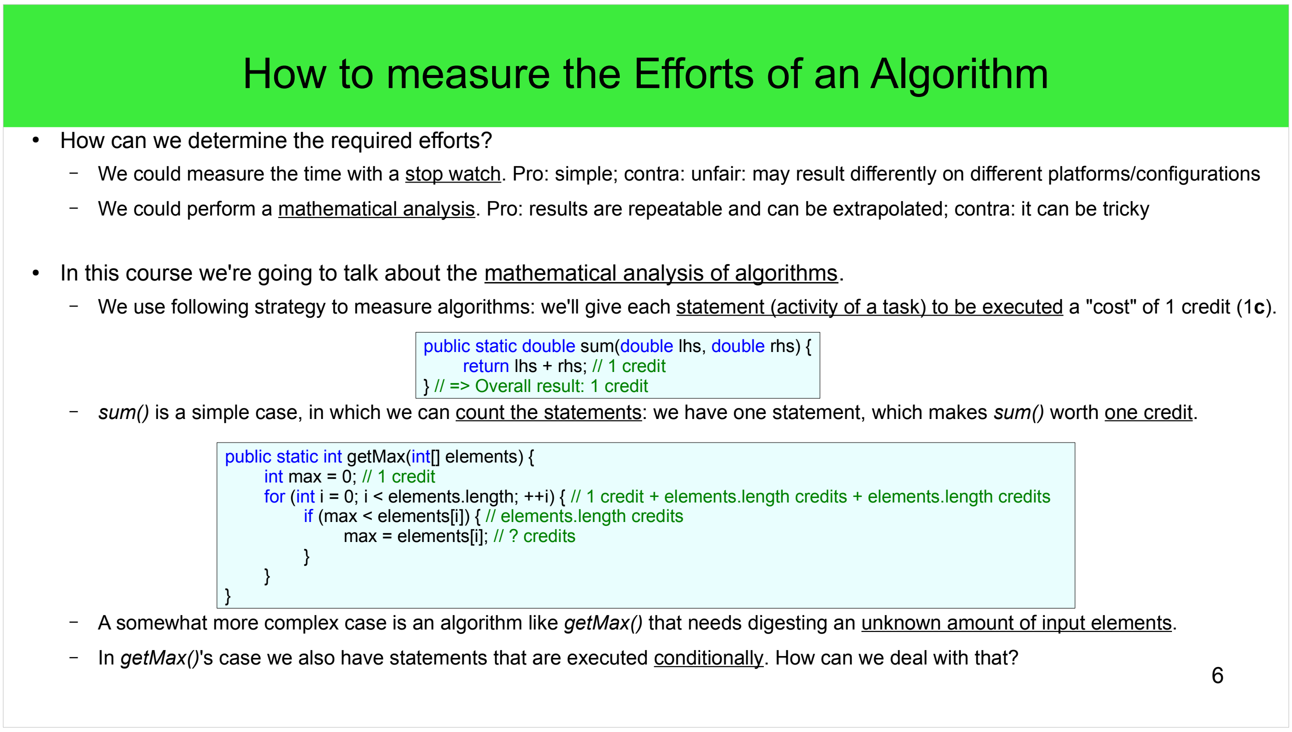
The Efforts of executing Algorithms

- We begin with measuring the efforts required to solve a problem (not necessarily an algorithm). These efforts could be:
  - Memory or space or distance
  - Time

- Example: How to count the amount of cars on the parking lot?
  - We could just count them.
  - We could select an area, count the cars and multiply by *nAreas* (i.e. the count of areas). This approach is called "sampling".
  - Following questions are relevant: How long will it take? What is required to perform the algorithm? How accurate is the result?

- In the realm of collections we analyze the efforts of the available operations on collections.
  - These operations are algorithms to solve problems like finding or accessing elements, inserting, appending and prepending elements.

- Knowing the efforts of specific operations on specific collections allows picking the most efficient approach for a task.
  - This is an essential skill for programmers, when we talk about programming tasks!
  - But for now we have to understand how algorithms are measured in practice.



How to measure the Efforts of an Algorithm

- How can we determine the required efforts?
  - We could measure the time with a stop watch. Pro: simple; contra: unfair: may result differently on different platforms/configurations
  - We could perform a mathematical analysis. Pro: results are repeatable and can be extrapolated; contra: it can be tricky

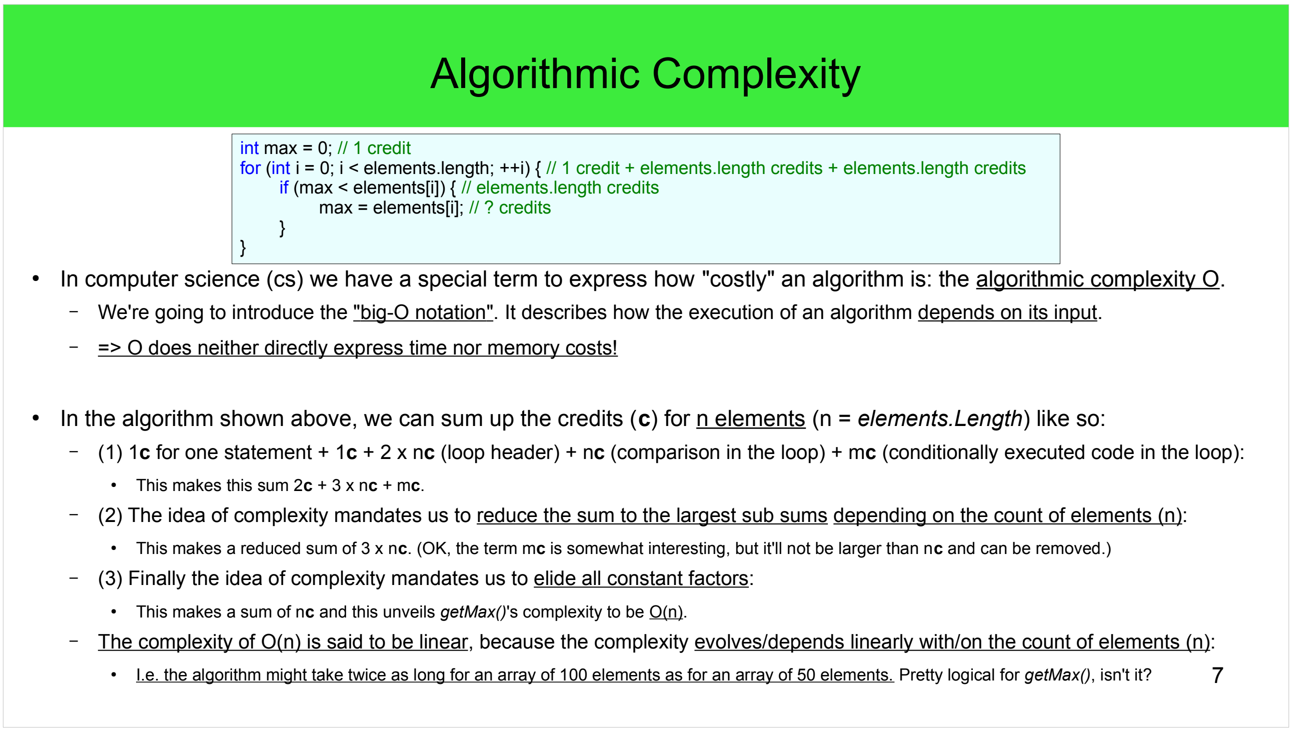
- In this course we're going to talk about the mathematical analysis of algorithms.
  - We use following strategy to measure algorithms: we'll give each statement (activity of a task) to be executed a "cost" of 1 credit (1**c**).

```
public static double sum(double lhs, double rhs) {
        return lhs + rhs; // 1 credit
} // => Overall result: 1 credit
```

  - *sum()* is a simple case, in which we can count the statements: we have one statement, which makes *sum()* worth one credit.

```
public static int getMax(int[] elements) {
        int max = 0; // 1 credit
        for (int i = 0; i < elements.length; ++i) { // 1 credit + elements.length credits + elements.length credits
                if (max < elements[i]) { // elements.length credits
                        max = elements[i]; // ? credits
                }
        }
}
```

  - A somewhat more complex case is an algorithm like *getMax()* that needs digesting an unknown amount of input elements.
  - In *getMax()*'s case we also have statements that are executed conditionally. How can we deal with that?



Algorithmic Complexity

```
int max = 0; // 1 credit
for (int i = 0; i < elements.length; ++i) { // 1 credit + elements.length credits + elements.length credits
        if (max < elements[i]) { // elements.length credits
                max = elements[i]; // ? credits
        }
}
```

- In computer science (cs) we have a special term to express how "costly" an algorithm is: the algorithmic complexity O.
  - We're going to introduce the "big-O notation". It describes how the execution of an algorithm depends on its input.
  - => O does neither directly express time nor memory costs!

- In the algorithm shown above, we can sum up the credits (**c**) for n elements (n = *elements.Length*) like so:
  - (1) 1**c** for one statement + 1**c** + 2 x n**c** (loop header) + n**c** (comparison in the loop) + m**c** (conditionally executed code in the loop):
    - This makes this sum 2**c** + 3 x n**c** + m**c**.
  - (2) The idea of complexity mandates us to reduce the sum to the largest sub sums depending on the count of elements (n):
    - This makes a reduced sum of 3 x n**c**. (OK, the term m**c** is somewhat interesting, but it'll not be larger than n**c** and can be removed.)
  - (3) Finally the idea of complexity mandates us to elide all constant factors:
    - This makes a sum of n**c** and this unveils *getMax()*'s complexity to be O(n).
  - The complexity of O(n) is said to be linear, because the complexity evolves/depends linearly with/on the count of elements (n):
    - I.e. the algorithm might take twice as long for an array of 100 elements as for an array of 50 elements. Pretty logical for *getMax()*, isn't it?

7

- We'll use the so called "Landau symbol" O (for German "Ordnung" and named after the German mathematician Edmund Landau, sometimes also called Bachmann-Landau notation also honoring the co-inventor of this notation, namely the German mathematician Paul Bachmann) as the symbol for the algorithmic complexity (complexity). Virtually the Landau symbol O expresses only the upper bound of algorithmic costs (other Landau symbols exist to express other bounds). But in practice only the symbol O is used.

Algorithmic Complexity – Examples

```
int max = 0; // 1 credit
for (int i = 0; i < elements.length; ++i) { // 1 credit + elements.length credits + elements.length credits
        if (max < elements[i]) { // elements.length credits
                max = elements[i]; // ? credits
        }
}
```

- Some facts on how complexity is "calculated" and how the big-O notation is applied:
  - Only the biggest term (that one with the most frequently use of n) counts for O, coefficients etc. are eliminated.
  - O doesn't take noisy other statements into consideration, like initialization of variables or the cost of comparison etc.
  - Here some examples:

$$7n+4 \rightarrow O(n) \qquad \frac{3}{4}n^2+n \rightarrow O(n^2)$$

$$13n \rightarrow O(n) \qquad n+2^n \rightarrow O(2^n)$$

  - (The expression of complexity is not so much "correct" mathematics.)

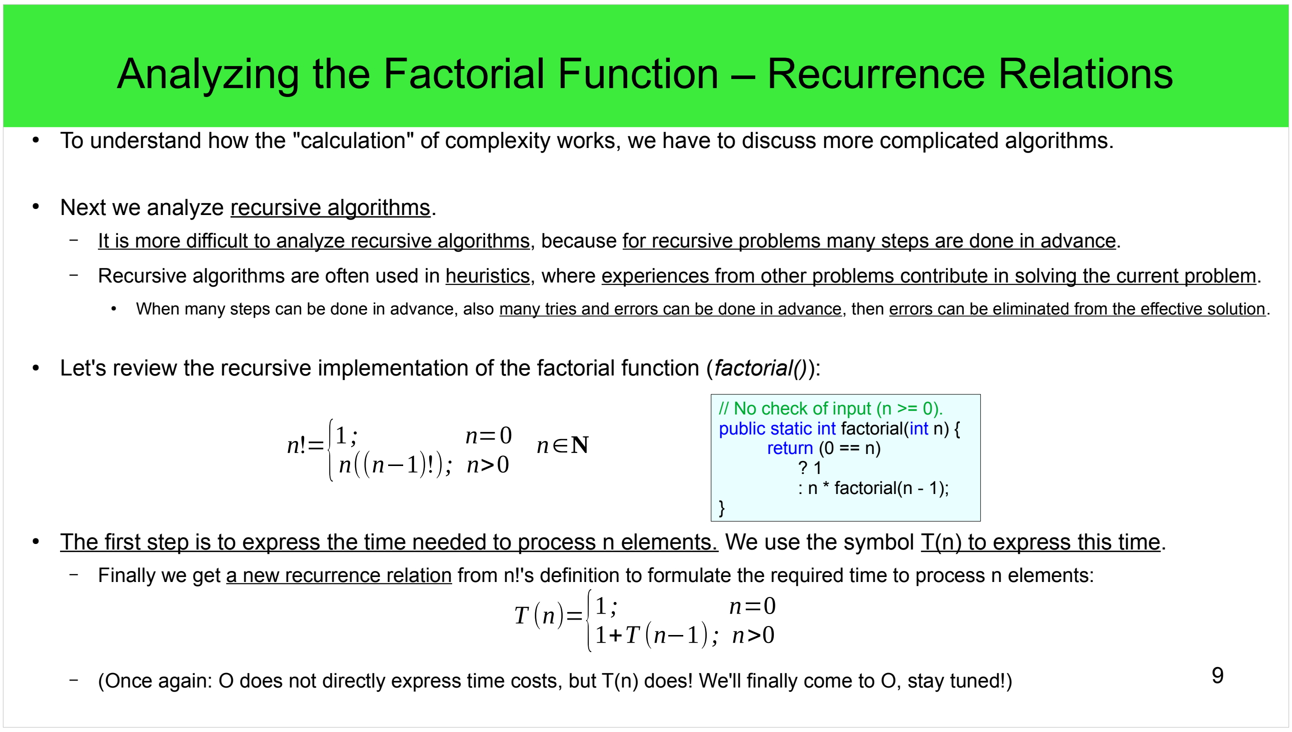
Notice
The O-notation as in 7n + 4 → O(n) means ***"7n + 4 does not significantly grow as fast as n".*** This "translation" of the notation shows, where "noise" is no part of the "equation".

- If an operation is completely independent from the (count of) elements it is said to have constant complexity.

```
int max = 0; // 1 credit -> independent of elements
```

$O(1)$

  - An algorithm having constant complexity is the most performant algorithm we can have basically.
  - (The absolutely most performant algorithm is the empty algorithm that has a complexity of O(0).)



Analyzing the Factorial Function – Recurrence Relations

- To understand how the "calculation" of complexity works, we have to discuss more complicated algorithms.

- Next we analyze recursive algorithms.
  - It is more difficult to analyze recursive algorithms, because for recursive problems many steps are done in advance.
  - Recursive algorithms are often used in heuristics, where experiences from other problems contribute in solving the current problem.
    - When many steps can be done in advance, also many tries and errors can be done in advance, then errors can be eliminated from the effective solution.

- Let's review the recursive implementation of the factorial function (*factorial()*):

$$n! = \begin{cases} 1; & n=0 \\ n((n-1)!); & n>0 \end{cases} \quad n \in \mathbf{N}$$

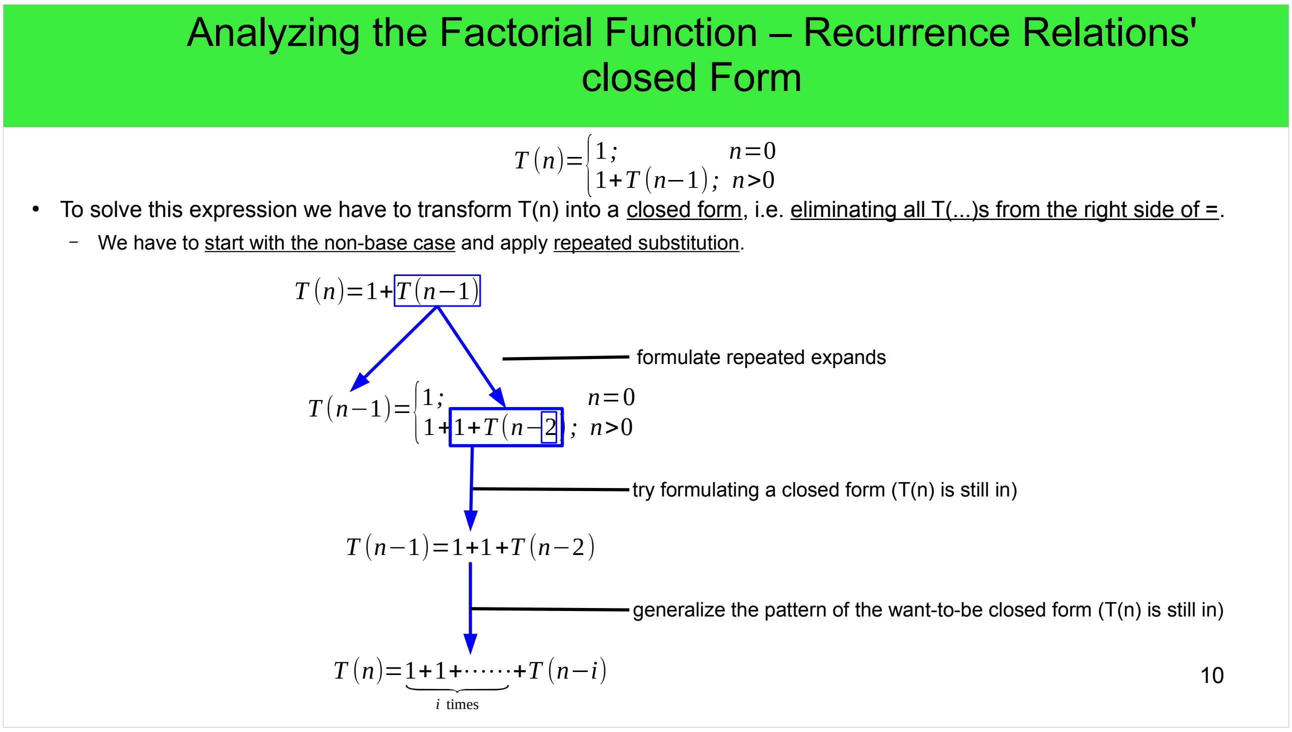
```
// No check of input (n >= 0).
public static int factorial(int n) {
        return (0 == n)
                ? 1
                : n * factorial(n - 1);
}
```

- The first step is to express the time needed to process n elements. We use the symbol T(n) to express this time.
  - Finally we get a new recurrence relation from n!'s definition to formulate the required time to process n elements:

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

  - (Once again: O does not directly express time costs, but T(n) does! We'll finally come to O, stay tuned!)

- The shown code for *factorial()* is not only recursive, it is also following the functional paradigm and language idiom (we used ?: instead of if/else statements).
- Recurrence relation: in German often (imprecisely) called "Differenzengleichung" or "Rekursionsgleichung".
- The recurrence relation of T looks almost exactly like the code of *factorial()*, but this is not the case for all types of algorithms.
- The elements in T:
  - Base case: the cost of 1 is a constant 1**c**.
  - Common case:
    - 1**c**: for evaluating n and the multiplication with the rest,
    - plus the time it takes to calculate T(n - 1).



Analyzing the Factorial Function – Recurrence Relations' closed Form

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

- To solve this expression we have to transform T(n) into a closed form, i.e. eliminating all T(...)s from the right side of =.
  - We have to start with the non-base case and apply repeated substitution.

$$T(n)=1+\boxed{T(n-1)}$$

formulate repeated expands

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

try formulating a closed form (T(n) is still in)

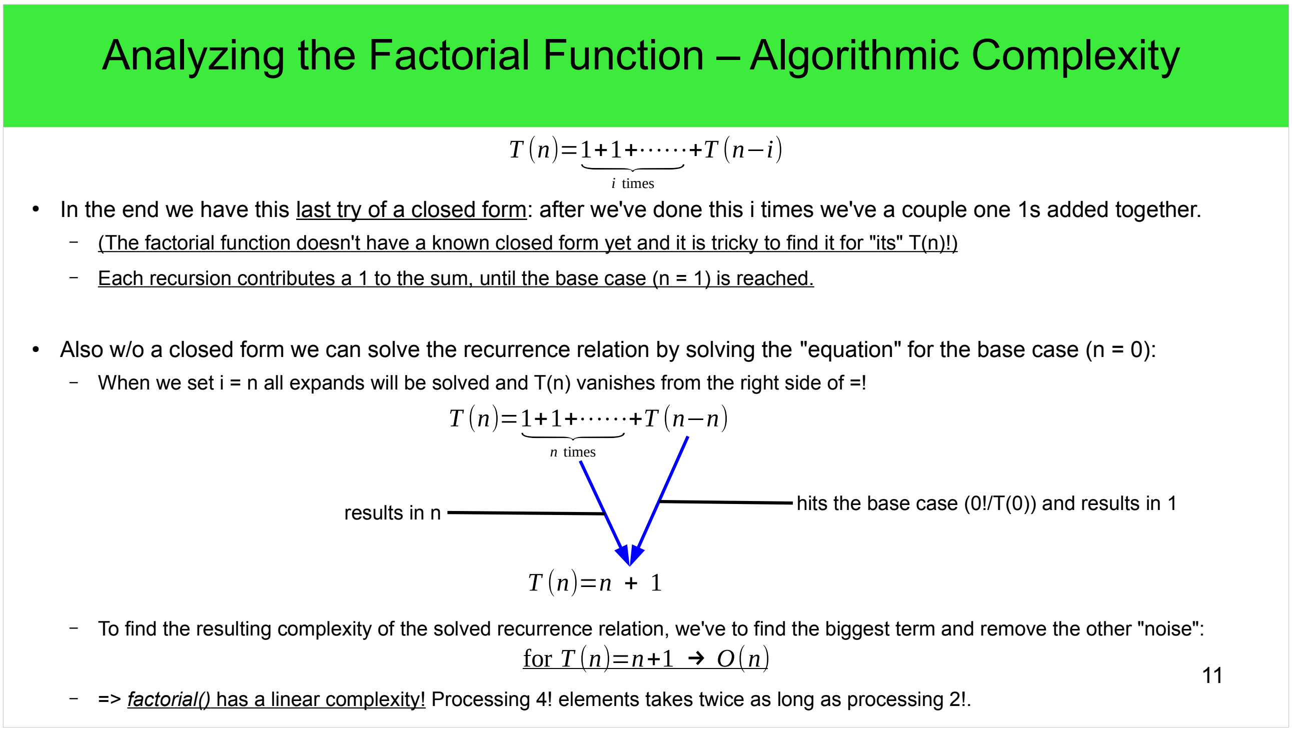
$$T(n-1)=1+1+T(n-2)$$

generalize the pattern of the want-to-be closed form (T(n) is still in)

$$T(n)=\underbrace{1+1+\cdots\cdots}_{i\text{ times}}+T(n-i)$$

10

- The development of the steps nicely show, that recursion means to pass a smaller version of the problem T(**n - 2**), which is smaller than the received **T(n - 1)**.
- In this example we can recognize that mathematical induction can be expressed with expanding recursions.
- A closed form of a function is basically expressing a function that has a deterministic count of steps independent from the input.
- The elements in T:
  - Base case: the cost of 1 is a constant 1**c**.
  - Common case:
    - 1**c**: for evaluating n and the multiplication with the rest,
    - plus the time it takes to calculate T(n - 1).



Analyzing the Factorial Function – Algorithmic Complexity

$$T(n)=\underbrace{1+1+\cdots\cdots}_{i\text{ times}}+T(n-i)$$

- In the end we have this last try of a closed form: after we've done this i times we've a couple one 1s added together.
  - (The factorial function doesn't have a known closed form yet and it is tricky to find it for "its" T(n)!)
  - Each recursion contributes a 1 to the sum, until the base case (n = 1) is reached.

- Also w/o a closed form we can solve the recurrence relation by solving the "equation" for the base case (n = 0):
  - When we set i = n all expands will be solved and T(n) vanishes from the right side of =!

$$T(n)=\underbrace{1+1+\cdots\cdots}_{n\text{ times}}+T(n-n)$$

results in n

hits the base case (0!/T(0)) and results in 1

$$T(n)=n\ +\ 1$$

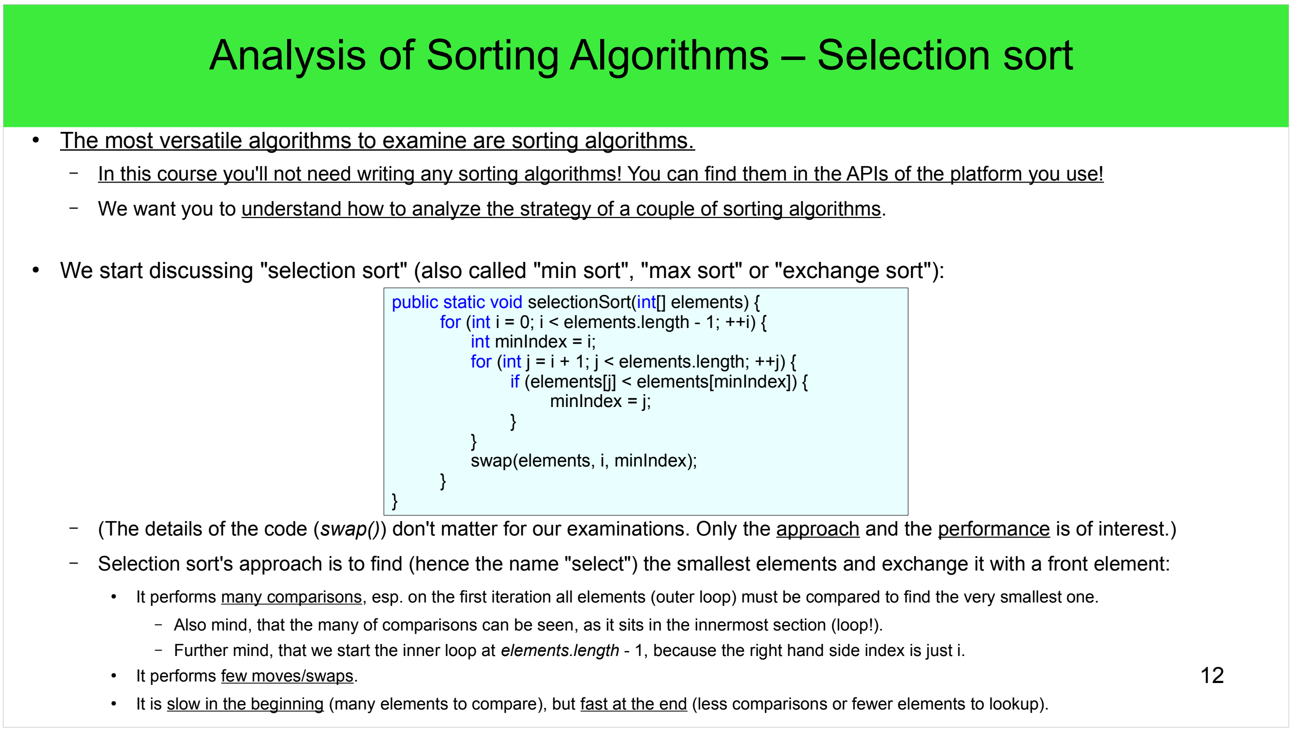
  - To find the resulting complexity of the solved recurrence relation, we've to find the biggest term and remove the other "noise":

$$\underline{\text{for } T(n)=n+1\ \rightarrow\ O(n)}$$

11

  - => *factorial()* has a linear complexity! Processing 4! elements takes twice as long as processing 2!.

- T(n) as n + 1 just means n multiplications plus the base case.
  - As a matter of fact nobody found the closed form for n! yet. There exist some asymptotic approaches, but no closed forms.
  - Mathematical functions can basically only have one of two forms: (1) closed form or (2) recursion
  - We could implement n! (i.e. really program in a programming language) using a table or map to cache results of past calculations! – But this is just an optimization.



Analysis of Sorting Algorithms – Selection sort

- The most versatile algorithms to examine are sorting algorithms.
  - In this course you'll not need writing any sorting algorithms! You can find them in the APIs of the platform you use!
  - We want you to understand how to analyze the strategy of a couple of sorting algorithms.

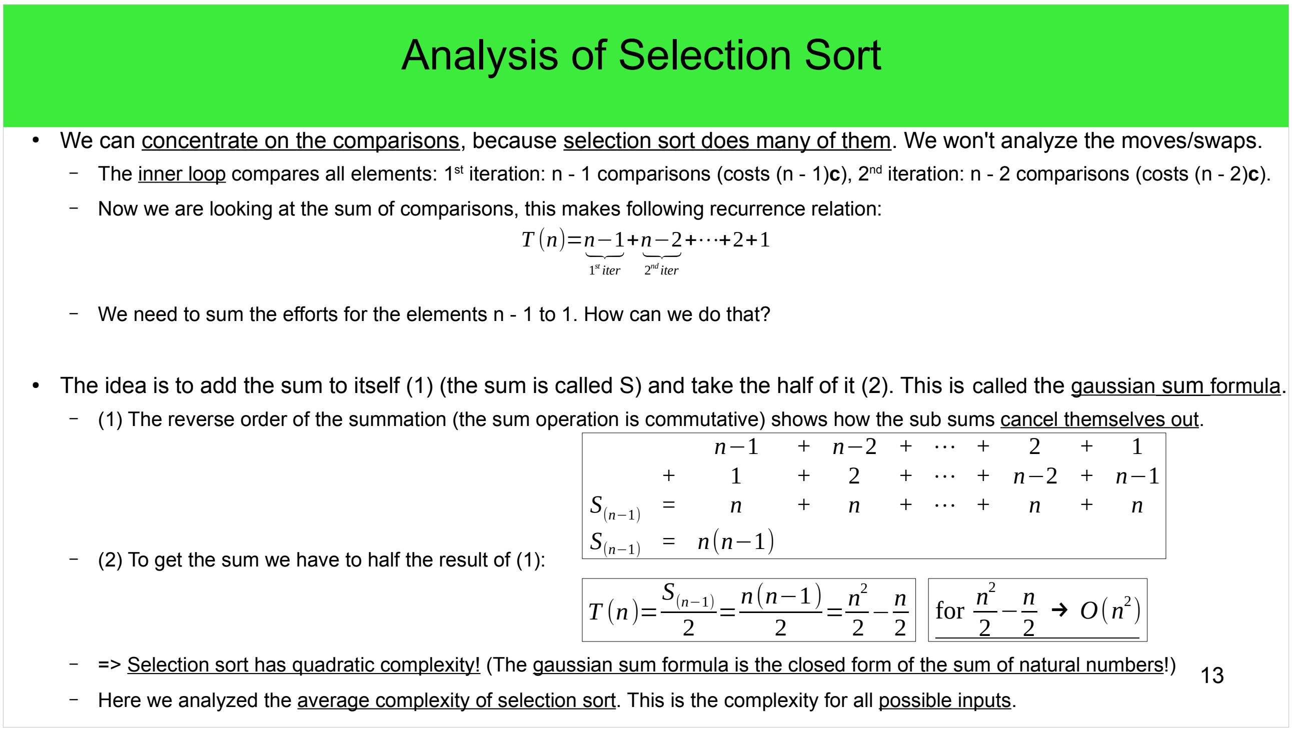
- We start discussing "selection sort" (also called "min sort", "max sort" or "exchange sort"):

```
public static void selectionSort(int[] elements) {
        for (int i = 0; i < elements.length - 1; ++i) {
              int minIndex = i;
              for (int j = i + 1; j < elements.length; ++j) {
                    if (elements[j] < elements[minIndex]) {
                          minIndex = j;
                    }
              }
              swap(elements, i, minIndex);
        }
}
```

  - (The details of the code (*swap()*) don't matter for our examinations. Only the approach and the performance is of interest.)
  - Selection sort's approach is to find (hence the name "select") the smallest elements and exchange it with a front element:
    - It performs many comparisons, esp. on the first iteration all elements (outer loop) must be compared to find the very smallest one.
      - Also mind, that the many of comparisons can be seen, as it sits in the innermost section (loop!).
      - Further mind, that we start the inner loop at *elements.length* - 1, because the right hand side index is just i.
    - It performs few moves/swaps.
    - It is slow in the beginning (many elements to compare), but fast at the end (less comparisons or fewer elements to lookup).

12

- We iterate the whole collection to find the smallest, then we iterate the rest (n - 1) until we find the smallest and exchange it with the 2nd element and so forth.
- We're not going to analyze "bubble sort"!
  - Approach: compare all neighbor elements and swap them in multiple iterations until all elements are sorted. The smaller (or larger) elements "bubble" to the top.
  - Selection sort is similarly inefficient but simpler to understand, because it sorts like humans would sort manually.



Analysis of Selection Sort

- We can concentrate on the comparisons, because selection sort does many of them. We won't analyze the moves/swaps.
  - The inner loop compares all elements: 1st iteration: n - 1 comparisons (costs (n - 1)**c**), 2nd iteration: n - 2 comparisons (costs (n - 2)**c**).
  - Now we are looking at the sum of comparisons, this makes following recurrence relation:

$$T(n)=\underbrace{n-1}_{1^{st}\,iter}+\underbrace{n-2}_{2^{nd}\,iter}+\cdots+2+1$$

  - We need to sum the efforts for the elements n - 1 to 1. How can we do that?

- The idea is to add the sum to itself (1) (the sum is called S) and take the half of it (2). This is called the gaussian sum formula.
  - (1) The reverse order of the summation (the sum operation is commutative) shows how the sub sums cancel themselves out.

$$\boxed{\begin{array}{llccccccccc} & & n-1 & + & n-2 & + & \cdots & + & 2 & + & 1 \\ & + & 1 & + & 2 & + & \cdots & + & n-2 & + & n-1 \\ S_{(n-1)} & = & n & + & n & + & \cdots & + & n & + & n \\ S_{(n-1)} & = & n(n-1) & & & & & & & & \end{array}}$$

  - (2) To get the sum we have to half the result of (1):

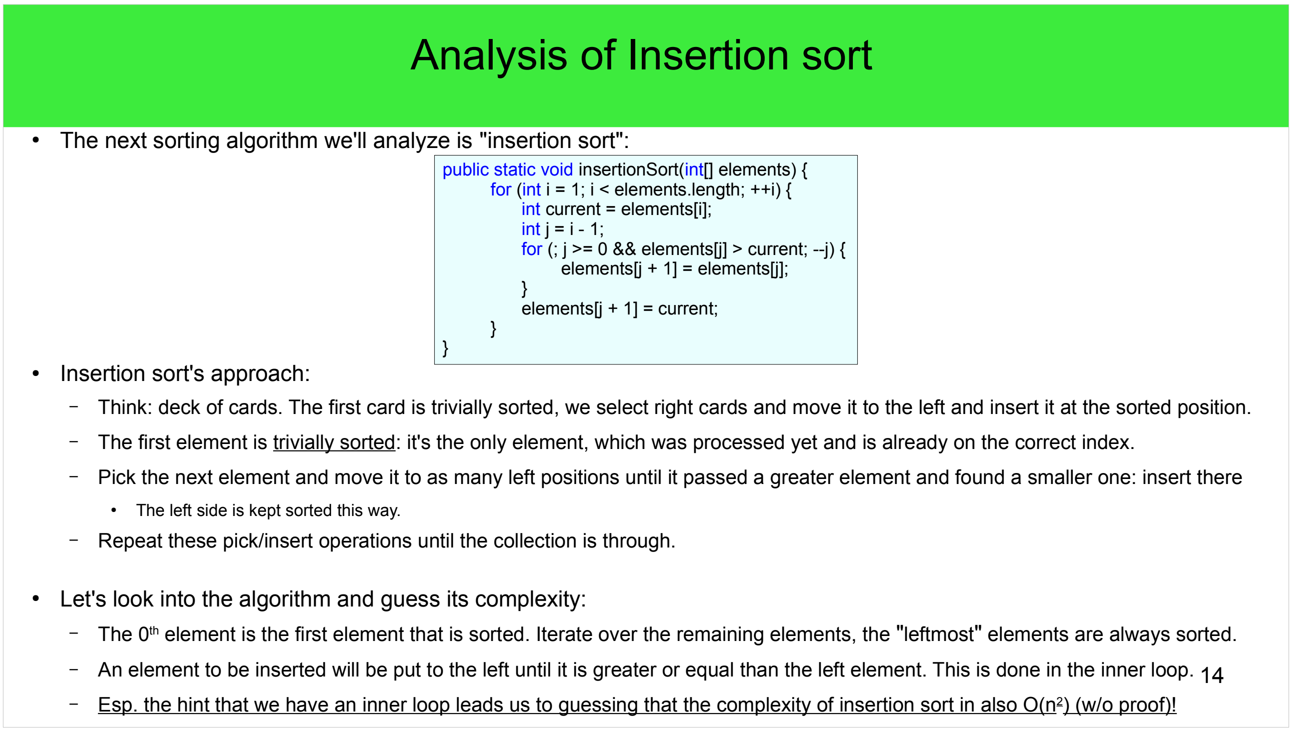
$$\boxed{T(n)=\frac{S_{(n-1)}}{2}=\frac{n(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}} \quad \boxed{\underline{\text{for } \frac{n^2}{2}-\frac{n}{2} \rightarrow O(n^2)}}$$

  - => Selection sort has quadratic complexity! (The gaussian sum formula is the closed form of the sum of natural numbers!)
  - Here we analyzed the average complexity of selection sort. This is the complexity for all possible inputs.

- We're somewhat off the "official" gaussian sum formula (n(n+1)/2) because we started at n - 1. → The inner loop is executed for (*elements.length* - 1) times, the outer loop for *elements.length* times! The original gaussian sum formula officially looks like this:

$$\begin{array}{llccccccccc} & & n & + & n-1 & + & \cdots & + & 2 & + & 1 \\ & + & 1 & + & 2 & + & \cdots & + & n-1 & + & n \\ S_{(n)} & = & n+1 & + & n+1 & + & \cdots & + & n+1 & + & n+1 \\ S_{(n)} & = & n(n+1) & & & & & & & & \end{array}$$

$$\boxed{\underline{=\frac{n(n+1)}{2}}}$$



Analysis of Insertion sort

- The next sorting algorithm we'll analyze is "insertion sort":

```
public static void insertionSort(int[] elements) {
        for (int i = 1; i < elements.length; ++i) {
             int current = elements[i];
             int j = i - 1;
             for (; j >= 0 && elements[j] > current; --j) {
                   elements[j + 1] = elements[j];
             }
             elements[j + 1] = current;
        }
}
```

- Insertion sort's approach:
  - Think: deck of cards. The first card is trivially sorted, we select right cards and move it to the left and insert it at the sorted position.
  - The first element is trivially sorted: it's the only element, which was processed yet and is already on the correct index.
  - Pick the next element and move it to as many left positions until it passed a greater element and found a smaller one: insert there
    - The left side is kept sorted this way.
  - Repeat these pick/insert operations until the collection is through.

- Let's look into the algorithm and guess its complexity:
  - The 0th element is the first element that is sorted. Iterate over the remaining elements, the "leftmost" elements are always sorted.
  - An element to be inserted will be put to the left until it is greater or equal than the left element. This is done in the inner loop.
  - Esp. the hint that we have an inner loop leads us to guessing that the complexity of insertion sort in also $O(n^2)$ (w/o proof)!

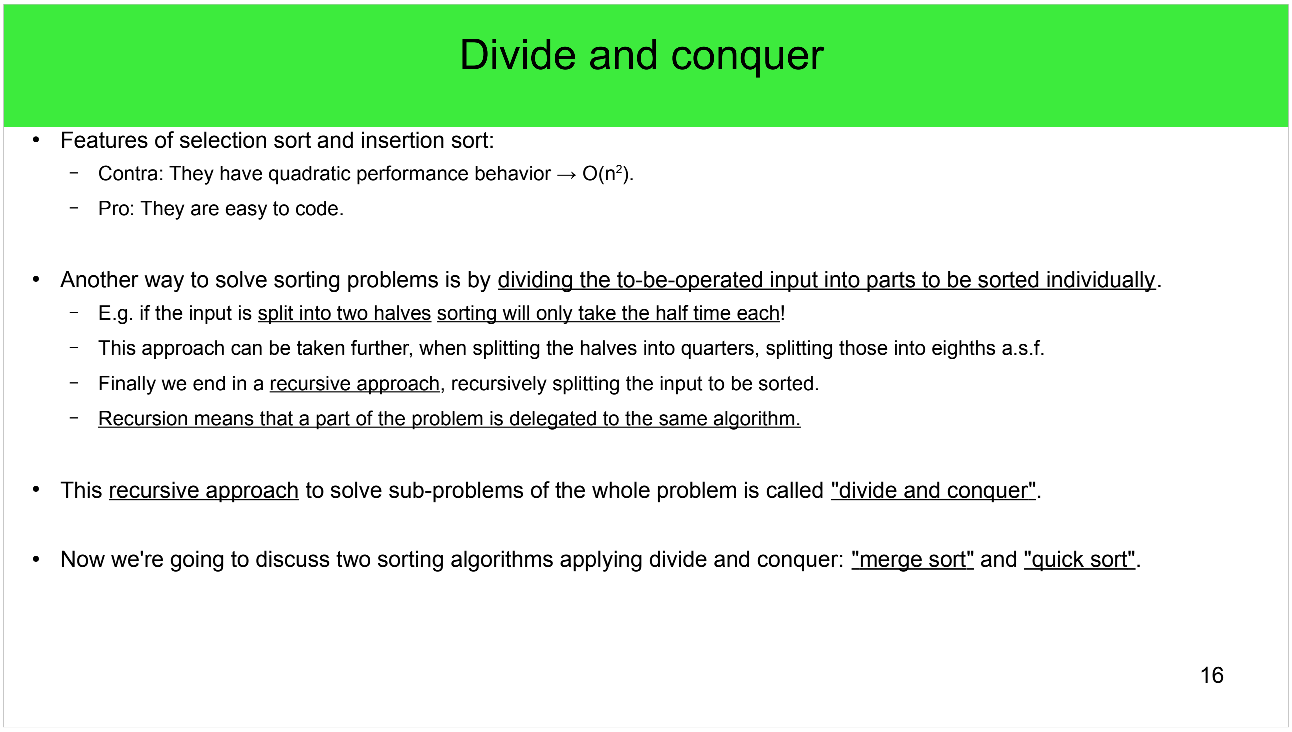
- In opposite to selection sort, we pick an element and take it with us until we find the first largest one, i.e. the comparison is done while we move. In selection sort we make the exact selection of the smallest element first.
- Nested loops are a marker for quadratic complexity.
  - The gaussian sum as we derived it our example (n(n-1)/2)) can also be used to calculate the count of clinks we here, when every person clinks glasses with every other person. Mind how the clinking activity is just processed as nested loop and how the quadratic behavior comes to be!

Selection sort vs Insertion sort

- Insertion sort:
  - It is fast in the beginning of sorting, because few elements need to be moved.
  - Slows down at the end of sorting, because potentially many moves must be performed.
  - The complexity is in the sum of moves/swaps, it performs only few comparisons.
  - The inner loop will compare all elements on the left (mind: those are already sorted).
  - The first element: one comparison, the second element: two comparisons … for the last: element n - 1 comparisons.
  - Best case: all elements are sorted! → $O(n)$, worst case: all elements are sorted in inverted order! → $O(n^2)$, average case: $O(n^2)$

- Selection sort (and bubble sort):
  - It is slow in the beginning of sorting, because many elements need to be compared to select the smallest element.
  - Gets faster at the end of the sorting, because few comparisons must be performed.
  - The complexity is in the sum of comparisons, it performs only few moves/swaps.
  - Best case: $O(n^2)$, worst case: $O(n^2)$, average case: $O(n^2)$ (Ouch!) → The performance is independent from the input!

- To choose an algorithm take into consideration, whether (1) elements are "cheap" to move/copy or (2) "cheap" to compare.
  - Pointers are cheap to copy: selection sort; instances of a "heavy" UDT are cheap to compare: insertion sort.



Divide and conquer

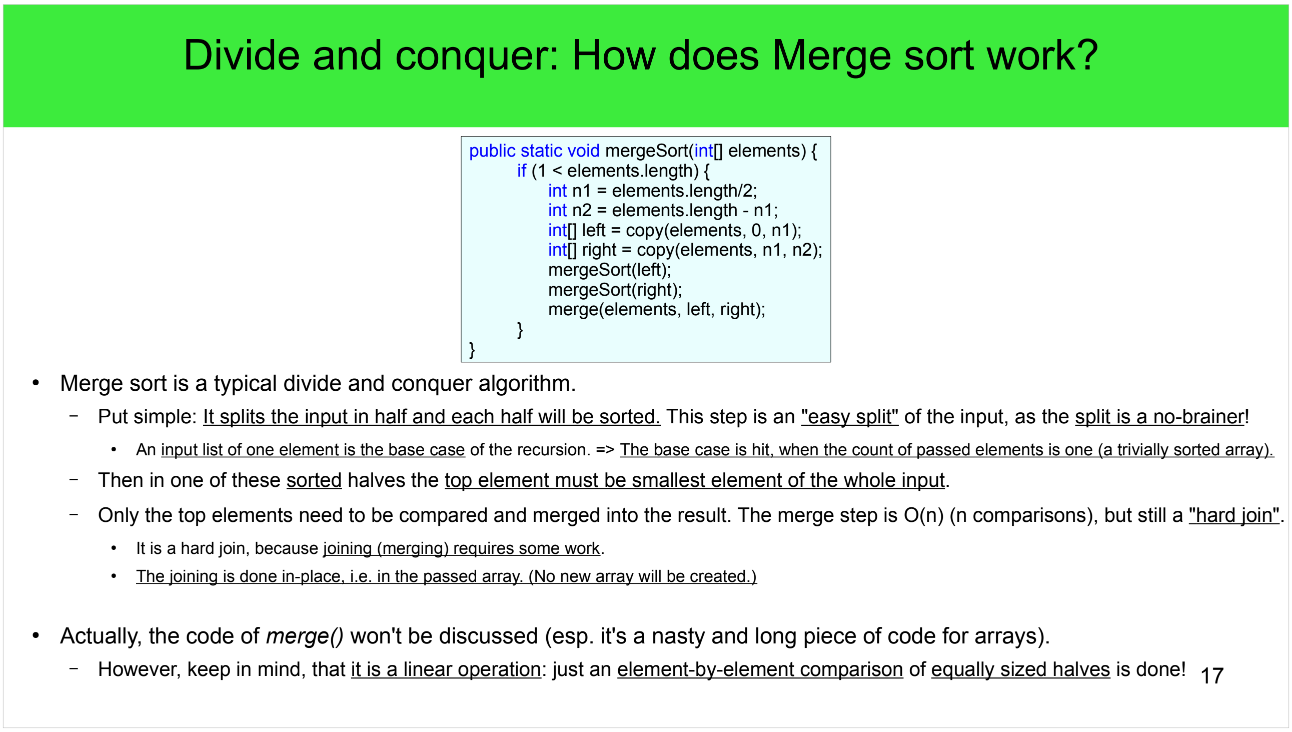
- Features of selection sort and insertion sort:
  - Contra: They have quadratic performance behavior → $O(n^2)$.
  - Pro: They are easy to code.

- Another way to solve sorting problems is by dividing the to-be-operated input into parts to be sorted individually.
  - E.g. if the input is split into two halves sorting will only take the half time each!
  - This approach can be taken further, when splitting the halves into quarters, splitting those into eighths a.s.f.
  - Finally we end in a recursive approach, recursively splitting the input to be sorted.
  - Recursion means that a part of the problem is delegated to the same algorithm.

- This recursive approach to solve sub-problems of the whole problem is called "divide and conquer".

- Now we're going to discuss two sorting algorithms applying divide and conquer: "merge sort" and "quick sort".

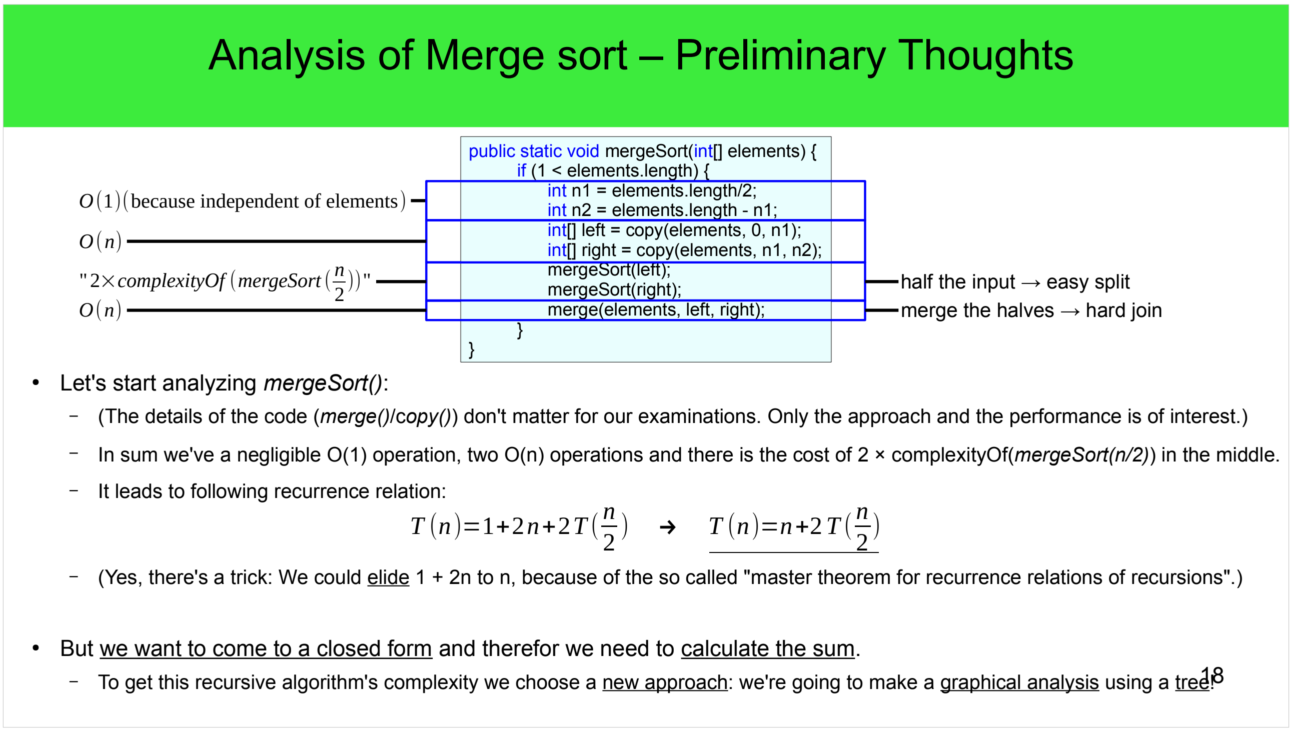
- Once again: The code of selection sort and insertion sort each has a hint for quadratic behavior: nested loops.



Divide and conquer: How does Merge sort work?

```
public static void mergeSort(int[] elements) {
        if (1 < elements.length) {
              int n1 = elements.length/2;
              int n2 = elements.length - n1;
              int[] left = copy(elements, 0, n1);
              int[] right = copy(elements, n1, n2);
              mergeSort(left);
              mergeSort(right);
              merge(elements, left, right);
        }
}
```

- Merge sort is a typical divide and conquer algorithm.
  - Put simple: It splits the input in half and each half will be sorted. This step is an "easy split" of the input, as the split is a no-brainer!
    - An input list of one element is the base case of the recursion. => The base case is hit, when the count of passed elements is one (a trivially sorted array).
  - Then in one of these sorted halves the top element must be smallest element of the whole input.
  - Only the top elements need to be compared and merged into the result. The merge step is O(n) (n comparisons), but still a "hard join".
    - It is a hard join, because joining (merging) requires some work.
    - The joining is done in-place, i.e. in the passed array. (No new array will be created.)
- Actually, the code of *merge()* won't be discussed (esp. it's a nasty and long piece of code for arrays).
  - However, keep in mind, that it is a linear operation: just an element-by-element comparison of equally sized halves is done!



Analysis of Merge sort – Preliminary Thoughts

```
public static void mergeSort(int[] elements) {
        if (1 < elements.length) {
                int n1 = elements.length/2;
                int n2 = elements.length - n1;
                int[] left = copy(elements, 0, n1);
                int[] right = copy(elements, n1, n2);
                mergeSort(left);
                mergeSort(right);
                merge(elements, left, right);
        }
}
```

$O(1)(\text{because independent of elements})$

$O(n)$

$"2\times complexityOf(mergeSort(\frac{n}{2}))"$

$O(n)$

half the input → easy split

merge the halves → hard join

- Let's start analyzing *mergeSort()*:
  - (The details of the code (*merge()*/*copy()*) don't matter for our examinations. Only the approach and the performance is of interest.)
  - In sum we've a negligible O(1) operation, two O(n) operations and there is the cost of 2 × complexityOf(*mergeSort(n/2)*) in the middle.
  - It leads to following recurrence relation:

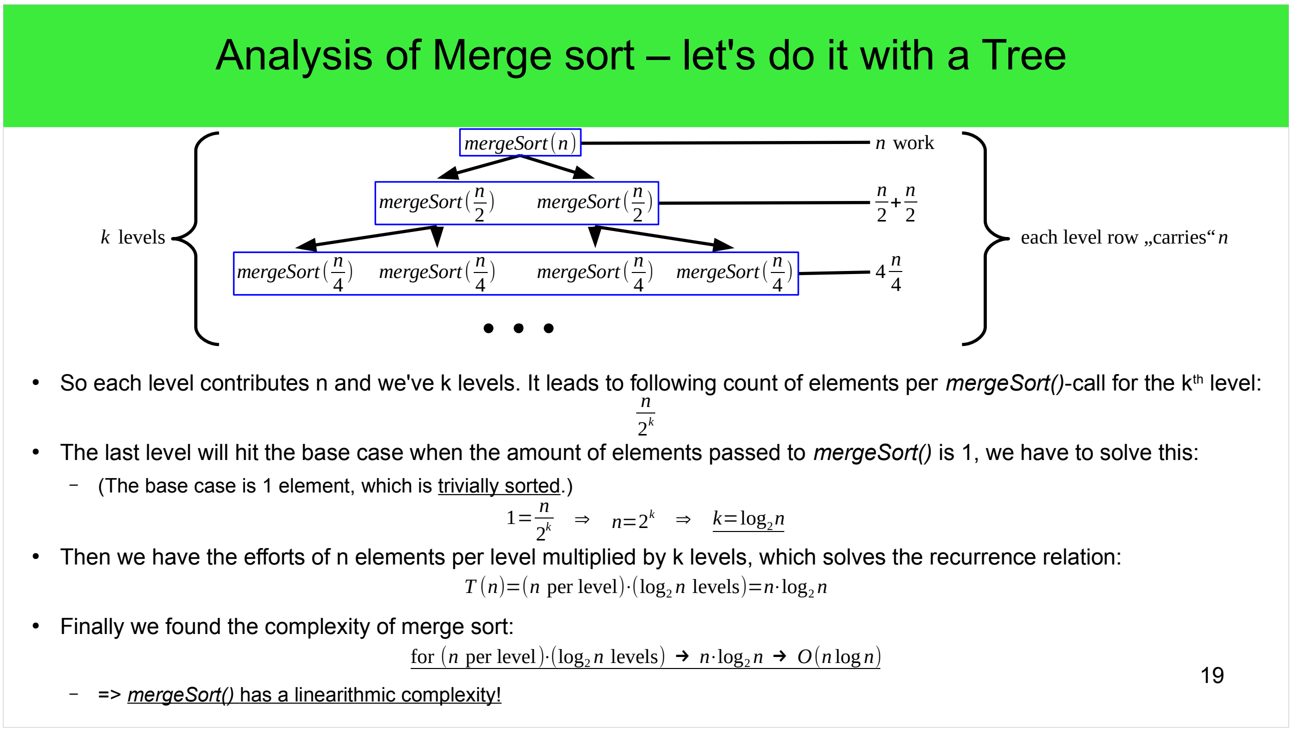
$$T(n)=1+2n+2T\left(\frac{n}{2}\right) \quad \rightarrow \quad \underline{T(n)=n+2T\left(\frac{n}{2}\right)}$$

  - (Yes, there's a trick: We could elide 1 + 2n to n, because of the so called "master theorem for recurrence relations of recursions".)

- But we want to come to a closed form and therefor we need to calculate the sum.
  - To get this recursive algorithm's complexity we choose a new approach: we're going to make a graphical analysis using a tree!

18

- Why is *merge()*'s complexity O(n) and not O(n/2)? It is because each comparison must be done against the effective result (which is n elements), while the elements of each half are put into the result.
- The recursion makes the sorting actually invisible, it just compares and sets one-to-two-element-arrays in the base case in the *merge()*-step.



Analysis of Merge sort – let's do it with a Tree

$mergeSort(n)$
n work
$mergeSort(\frac{n}{2})$ $mergeSort(\frac{n}{2})$
$\frac{n}{2}+\frac{n}{2}$
k levels
each level row „carries“ n
$mergeSort(\frac{n}{4})$ $mergeSort(\frac{n}{4})$ $mergeSort(\frac{n}{4})$ $mergeSort(\frac{n}{4})$
$4\frac{n}{4}$
• • •

- So each level contributes n and we've k levels. It leads to following count of elements per *mergeSort()*-call for the kth level:

$$\frac{n}{2^k}$$

- The last level will hit the base case when the amount of elements passed to *mergeSort()* is 1, we have to solve this:
  - (The base case is 1 element, which is trivially sorted.)

$$1=\frac{n}{2^k} \quad \Rightarrow \quad n=2^k \quad \Rightarrow \quad \underline{k=\log_2 n}$$

- Then we have the efforts of n elements per level multiplied by k levels, which solves the recurrence relation:

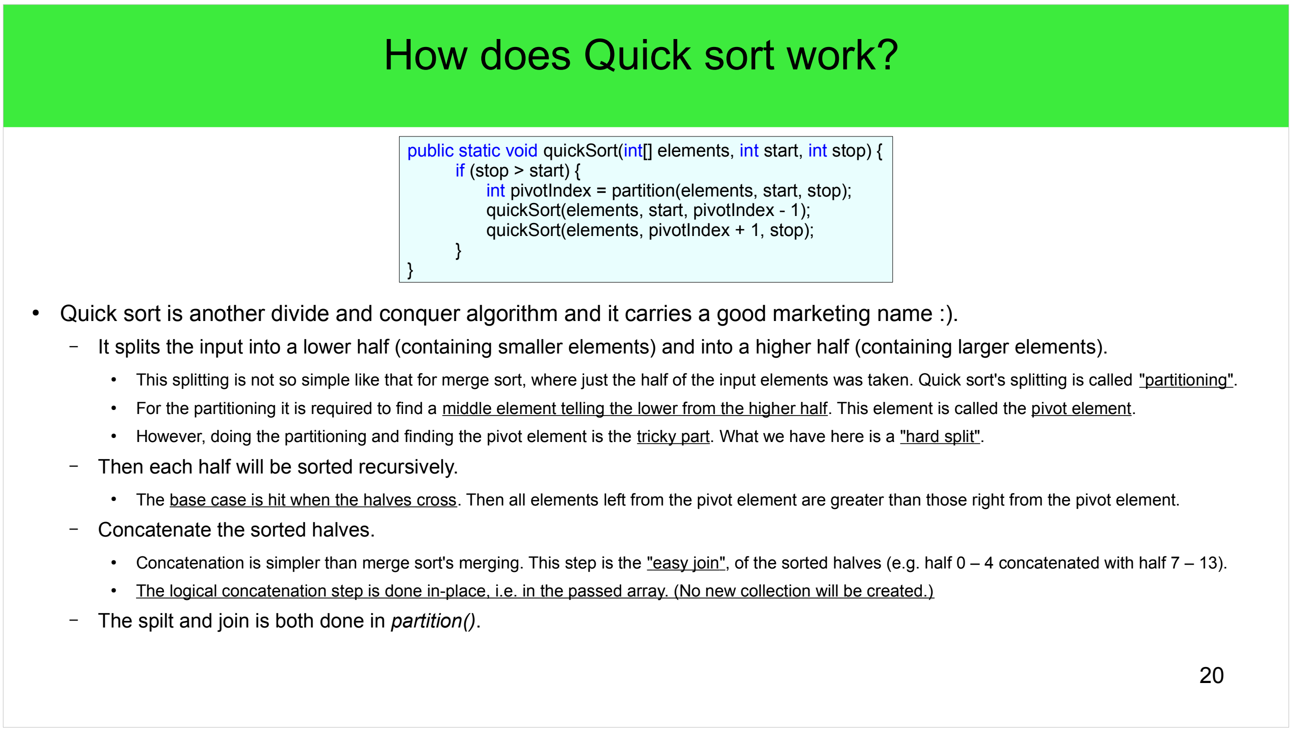
$$T(n)=(n \text{ per level})\cdot(\log_2 n \text{ levels})=n\cdot\log_2 n$$

- Finally we found the complexity of merge sort:

$$\underline{\text{for } (n \text{ per level})\cdot(\log_2 n \text{ levels}) \rightarrow n\cdot\log_2 n \rightarrow O(n\log n)}$$

  - => *mergeSort()* has a linearithmic complexity!

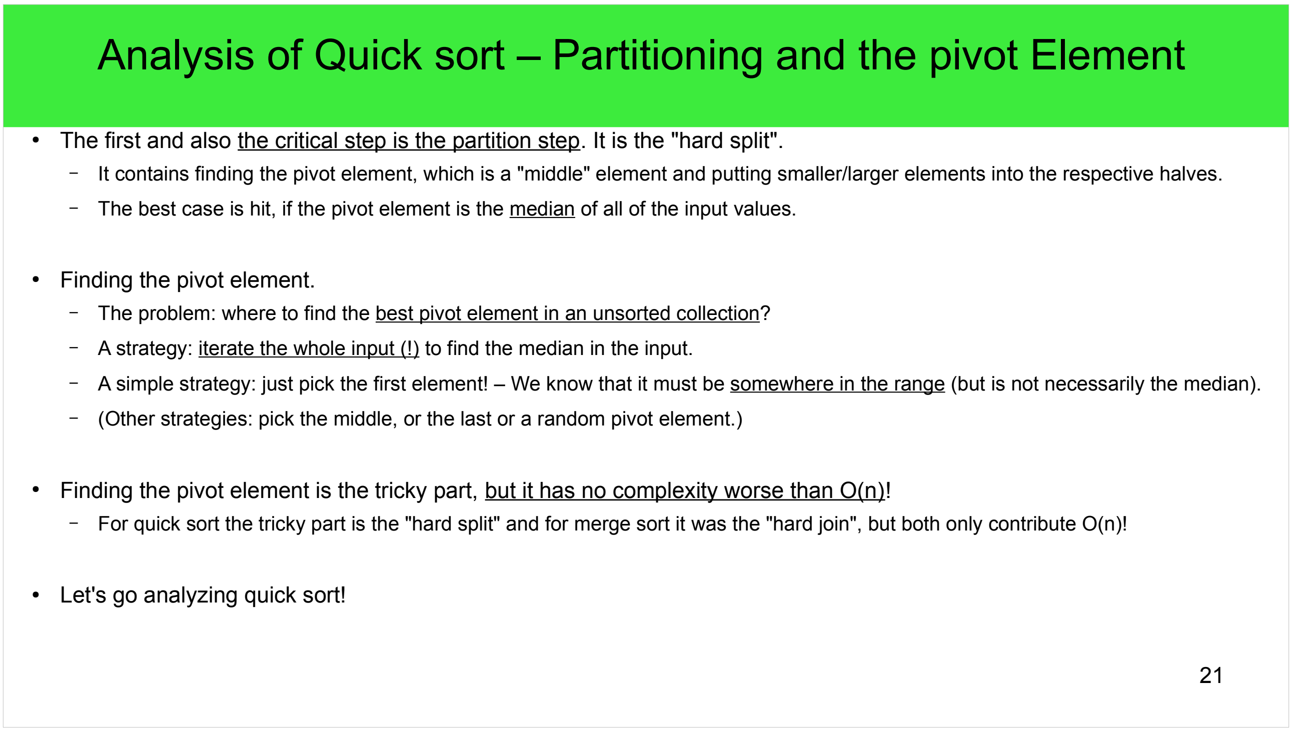
- The question we want to answer here: how many levels do we have to step down, until the base case is hit?
- Actually, the n in the recurrence relation T(n) = n + 2T(n/2) dissolved in the analysis as well, because it is just also halved, quartered etc. during the recursive steps.
- The word "linearithmic" is a portmanteau of the words "linear" and "logarithmic".



How does Quick sort work?

```
public static void quickSort(int[] elements, int start, int stop) {
        if (stop > start) {
             int pivotIndex = partition(elements, start, stop);
             quickSort(elements, start, pivotIndex - 1);
             quickSort(elements, pivotIndex + 1, stop);
        }
}
```

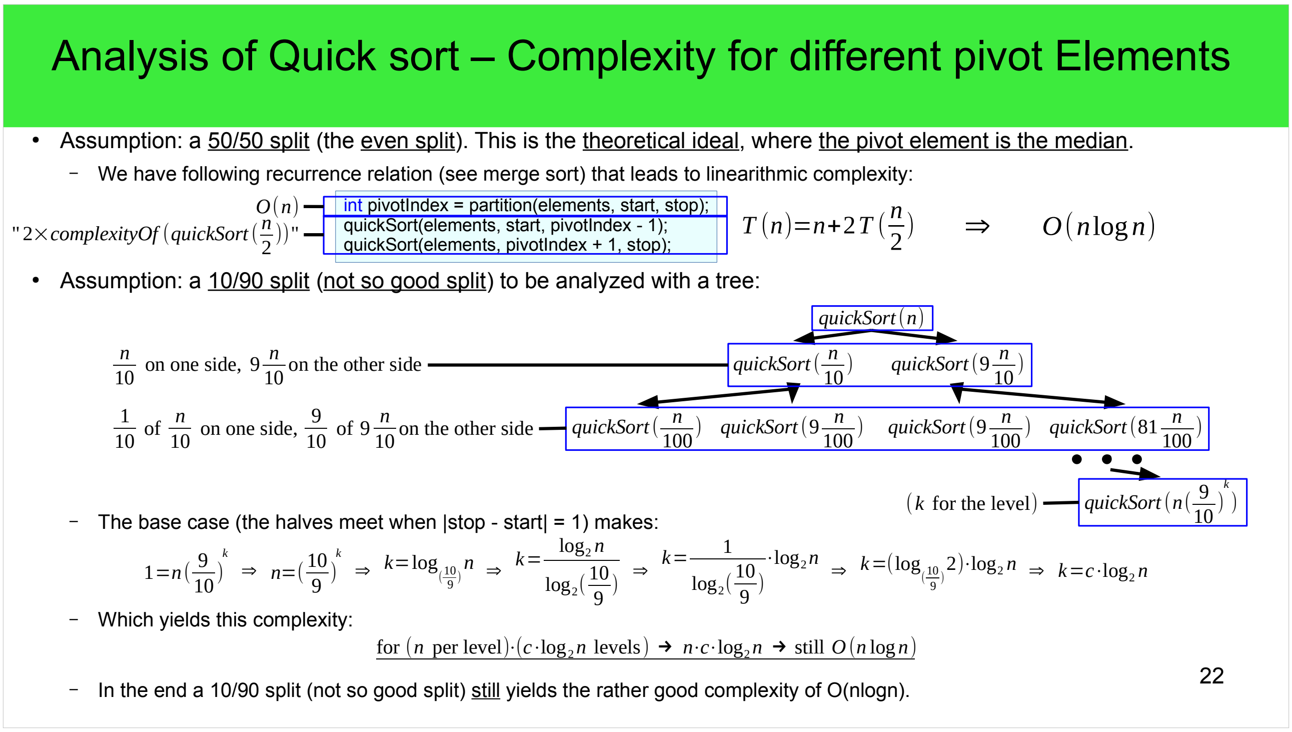
- Quick sort is another divide and conquer algorithm and it carries a good marketing name :).
  - It splits the input into a lower half (containing smaller elements) and into a higher half (containing larger elements).
    - This splitting is not so simple like that for merge sort, where just the half of the input elements was taken. Quick sort's splitting is called "partitioning".
    - For the partitioning it is required to find a middle element telling the lower from the higher half. This element is called the pivot element.
    - However, doing the partitioning and finding the pivot element is the tricky part. What we have here is a "hard split".
  - Then each half will be sorted recursively.
    - The base case is hit when the halves cross. Then all elements left from the pivot element are greater than those right from the pivot element.
  - Concatenate the sorted halves.
    - Concatenation is simpler than merge sort's merging. This step is the "easy join", of the sorted halves (e.g. half 0 – 4 concatenated with half 7 – 13).
    - The logical concatenation step is done in-place, i.e. in the passed array. (No new collection will be created.)
  - The spilt and join is both done in *partition()*.



Analysis of Quick sort – Partitioning and the pivot Element

- The first and also the critical step is the partition step. It is the "hard split".
  - It contains finding the pivot element, which is a "middle" element and putting smaller/larger elements into the respective halves.
  - The best case is hit, if the pivot element is the median of all of the input values.
- Finding the pivot element.
  - The problem: where to find the best pivot element in an unsorted collection?
  - A strategy: iterate the whole input (!) to find the median in the input.
  - A simple strategy: just pick the first element! – We know that it must be somewhere in the range (but is not necessarily the median).
  - (Other strategies: pick the middle, or the last or a random pivot element.)
- Finding the pivot element is the tricky part, but it has no complexity worse than O(n)!
  - For quick sort the tricky part is the "hard split" and for merge sort it was the "hard join", but both only contribute O(n)!
- Let's go analyzing quick sort!

- The difference to merge sort is, that the pivot is an explicitly chosen element, of which we know, on which side "the only larger" and the only smaller ones will be put on, then concatenating on the pivot is trivial.



Analysis of Quick sort – Complexity for different pivot Elements

- Assumption: a 50/50 split (the even split). This is the theoretical ideal, where the pivot element is the median.
  - We have following recurrence relation (see merge sort) that leads to linearithmic complexity:

$O(n)$

$"2\times complexityOf(quickSort(\frac{n}{2}))"$

```
int pivotIndex = partition(elements, start, stop);
quickSort(elements, start, pivotIndex - 1);
quickSort(elements, pivotIndex + 1, stop);
```

$$T(n)=n+2T(\frac{n}{2}) \quad \Rightarrow \quad O(n\log n)$$

- Assumption: a 10/90 split (not so good split) to be analyzed with a tree:

$quickSort(n)$

$\frac{n}{10}$ on one side, $9\frac{n}{10}$ on the other side

$quickSort(\frac{n}{10})$ $quickSort(9\frac{n}{10})$

$\frac{1}{10}$ of $\frac{n}{10}$ on one side, $\frac{9}{10}$ of $9\frac{n}{10}$ on the other side

$quickSort(\frac{n}{100})$ $quickSort(9\frac{n}{100})$ $quickSort(9\frac{n}{100})$ $quickSort(81\frac{n}{100})$

• • •

$(k$ for the level$)$ $quickSort(n(\frac{9}{10})^k)$

  - The base case (the halves meet when |stop - start| = 1) makes:

$$1=n(\frac{9}{10})^k \Rightarrow n=(\frac{10}{9})^k \Rightarrow k=\log_{(\frac{10}{9})}n \Rightarrow k=\frac{\log_2 n}{\log_2(\frac{10}{9})} \Rightarrow k=\frac{1}{\log_2(\frac{10}{9})}\cdot\log_2 n \Rightarrow k=(\log_{(\frac{10}{9})}2)\cdot\log_2 n \Rightarrow k=c\cdot\log_2 n$$

  - Which yields this complexity:

$$\underline{\text{for } (n \text{ per level})\cdot(c\cdot\log_2 n \text{ levels}) \rightarrow n\cdot c\cdot\log_2 n \rightarrow \text{still } O(n\log n)}$$

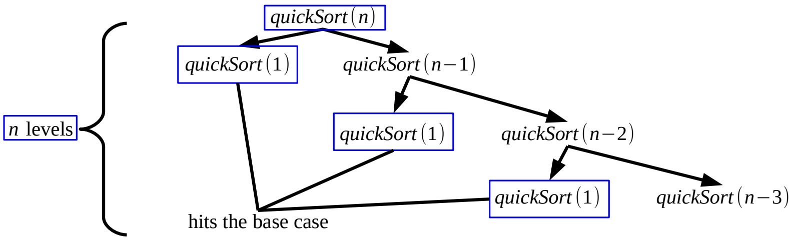
  - In the end a 10/90 split (not so good split) still yields the rather good complexity of O(nlogn).

$$Change\ base\ rule: \log_a b=\frac{\log_c b}{\log_c a}$$

$$Reciprocal\ rule: \frac{1}{\log_a b}=\log_b a$$

Analysis of Quick sort – The worst Case for a pivot Element

- The worst case is a 1/n-1 split. Here the pivot element is the smallest element and the input is already sorted.
  - The complexity of this worst case can be shown with a tree as well:



$quickSort(n)$

$quickSort(1)$

$quickSort(n-1)$

n levels

$quickSort(1)$

$quickSort(n-2)$

$quickSort(1)$

$quickSort(n-3)$

hits the base case

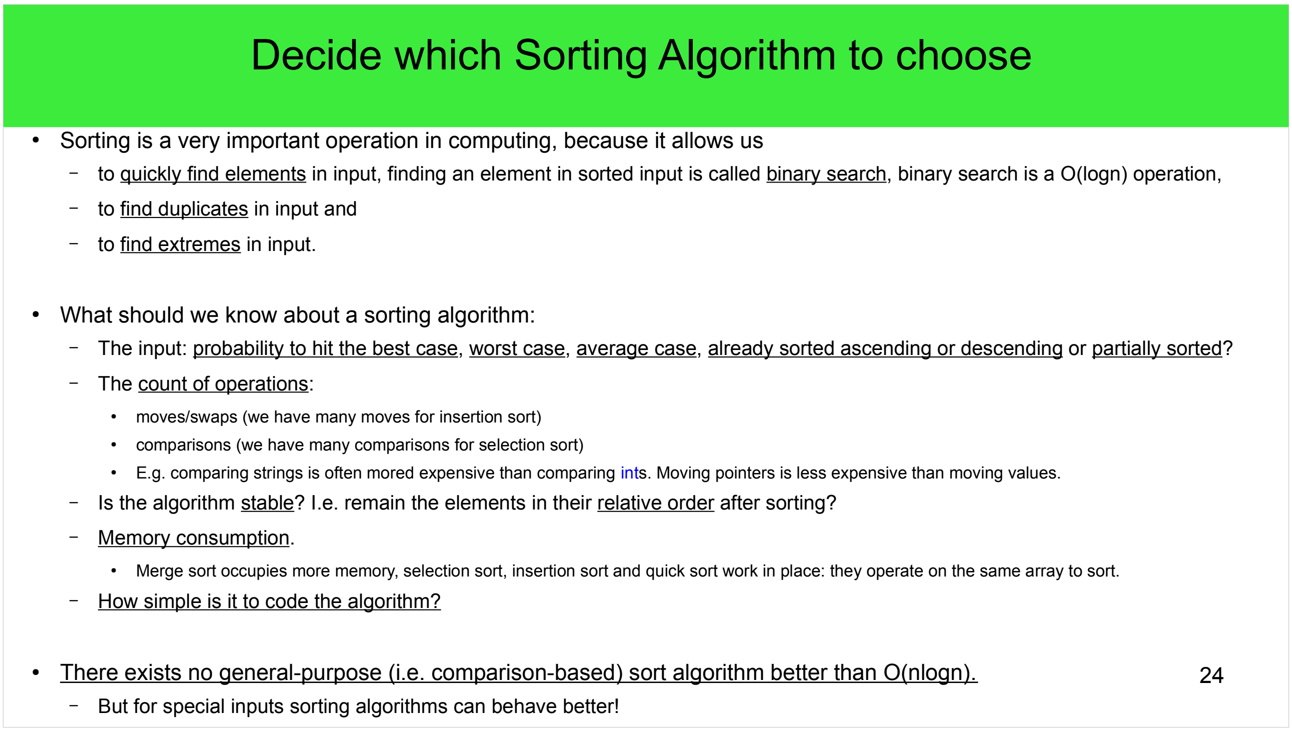
  - The worst case yields quadratic complexity:

$$\underline{\text{for } (n \text{ per level}) \cdot (n \text{ levels}) \rightarrow n \cdot n \rightarrow O(n^2)}$$

  - The effect is that we have many partitioning steps but sorting never takes place.
  - On each level the list to be sorted is just reduced by one element.

- Combinations of selecting an extreme pivot element and having an already sorted input make the worse case here!

- Virtually there is no ideal strategy to find a suitable pivot element to avoid the worst case.

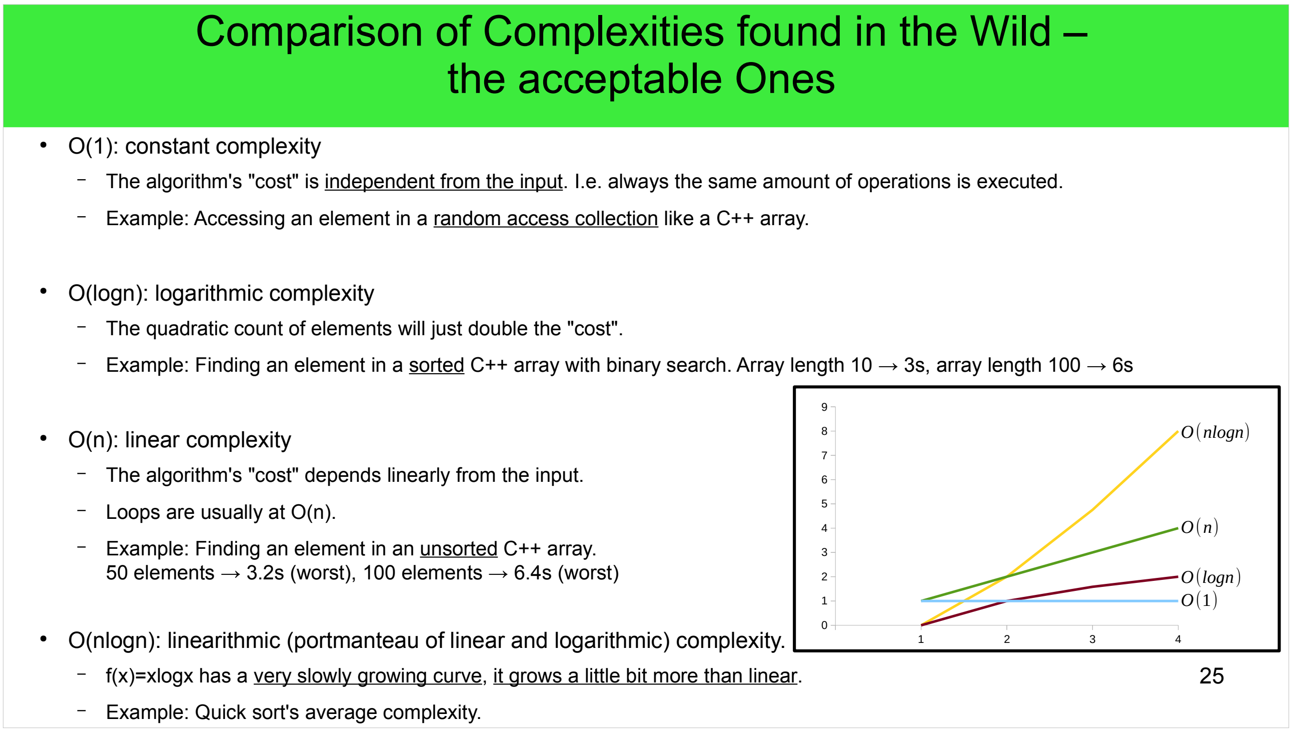


Decide which Sorting Algorithm to choose

- Sorting is a very important operation in computing, because it allows us
  - to quickly find elements in input, finding an element in sorted input is called binary search, binary search is a O(logn) operation,
  - to find duplicates in input and
  - to find extremes in input.
- What should we know about a sorting algorithm:
  - The input: probability to hit the best case, worst case, average case, already sorted ascending or descending or partially sorted?
  - The count of operations:
    - moves/swaps (we have many moves for insertion sort)
    - comparisons (we have many comparisons for selection sort)
    - E.g. comparing strings is often mored expensive than comparing ints. Moving pointers is less expensive than moving values.
  - Is the algorithm stable? I.e. remain the elements in their relative order after sorting?
  - Memory consumption.
    - Merge sort occupies more memory, selection sort, insertion sort and quick sort work in place: they operate on the same array to sort.
  - How simple is it to code the algorithm?
- There exists no general-purpose (i.e. comparison-based) sort algorithm better than O(nlogn).
  - But for special inputs sorting algorithms can behave better!

24

- Comparison:
  - Selection sort: always → O(n^2).
  - Insertion sort: best case (input is sorted) → O(n) (i.e. better than divide-and-conquer-based sorting), other cases → O(n^2)
  - Merge sort: always → O(nlogn) (i.e. better than quick sort in the average case)
  - Quick sort: worst case (input is sorted, pivot is smallest or largest element) → O(n^2), other cases → O(nlogn)
- Stable order sorting is esp. important for non-primitive type input collections, because non-primitive types have an identity. – In case they move their relative positions in the input, depending algorithms can break. For primitive types, the stability of sorting doesn't matter, we cannot tell two occasions of the int 42 in an array, if they are swapped, however, it won't influence any dependent algorithms. Some libraries, e.g. *Arrays.sort()* in some Java versions, use quick sort for primitive types, which is unstable, but merge sort for non-primitive types, because it is stable.
- A sort algorithm, that needs to create a new collection is usually significantly more expensive for primitive-type input collections, than for non-primitive type ones. Mind that doubling an int-array means to requires two time the input memory to operate! - For an array only holding references to non-primitive types, the doubling-requirement is not so much of an impact.
- Sort algorithms of popular libraries or system-level libraries often combine different algorithm depending on the input.
  - E.g. quicksort "costs" O(nlogn) and we just learned, that constant factors are removed from any complexity considerations, because they are independent of n (the "functions" are to consider for their limiting behavior in the complexity only). But practically, those constants can only be ignored, if n is not too small. If n is too small, the constants' impact can be significant for the real costs. Esp. the impact of constants for smaller ns can be more heavy weight for quicksort than for insertionsort, therefor it is thinkable, that a quick sort implementation can internally switch to insertion sort, when the recursion reaches to small partitions (small ns). These constant factors can be even more relevant, if a quicksort implementation creates threads/tasks to parallelize its work.



Comparison of Complexities found in the Wild – the acceptable Ones

- O(1): constant complexity
  - The algorithm's "cost" is independent from the input. I.e. always the same amount of operations is executed.
  - Example: Accessing an element in a random access collection like a C++ array.
- O(logn): logarithmic complexity
  - The quadratic count of elements will just double the "cost".
  - Example: Finding an element in a sorted C++ array with binary search. Array length 10 → 3s, array length 100 → 6s
- O(n): linear complexity
  - The algorithm's "cost" depends linearly from the input.
  - Loops are usually at O(n).
  - Example: Finding an element in an unsorted C++ array.
    50 elements → 3.2s (worst), 100 elements → 6.4s (worst)
- O(nlogn): linearithmic (portmanteau of linear and logarithmic) complexity.
  - f(x)=xlogx has a very slowly growing curve, it grows a little bit more than linear.
  - Example: Quick sort's average complexity.

Comparison of Complexities found in the Wild – the unacceptable Ones

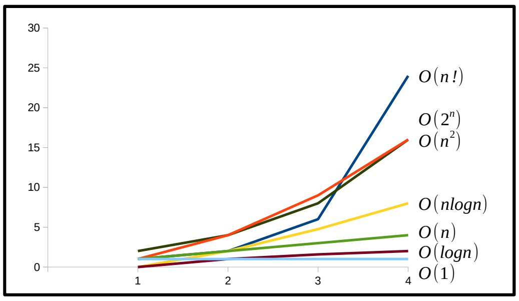
- $O(n^2)$: quadratic complexity
  - Double input → quadruple time, half input → quarter of time
  - Example: insertion sort. 50 elements → 2.4s, 100 elements → 9.6s, joining data in a table

- $O(c^n)$ (typically $O(2^n)$): exponential complexity
  - Towers of Hanoi ($O(2^n)$): Moving a tower of eleven discs will take twice as long as for ten discs (factor of two for each additional disc).
  - Examples: Towers of Hanoi, the Traveling Salesperson Problem (TSP)
  - "recursive calls over n while looping over c"

- O(n!): factorial complexity
  - "loop over n and recursive call in the loop over n – 1"

- Constant, logarithmic, linear and linearithmic complexity are acceptable "costs" for most cases in the wild.

- Quadratic, exponential and factorial complexity are typically unacceptable "costs" for most cases in the wild.
  - In other words: Algorithms "worse" than O(nlogn) are typically unacceptable for industry quality code!



Algorithms on Collections and Collections' Methods

```
public static int getMax(int[] elements) { // getMax() as static method
	int max = 0;                                   // operating on a passed collection:
	for (int i = 0; i < elements.length; ++i) {
		if (max < elements[i]) {
			max = elements[i];
		}
	}
}
```

```
public class List {
	public int getMax() { // getMax() as fictive method
		int max = 0;      // of a List managing ints:
		for (int i = 0; i < this.size(); ++i) {
			if (max < this.get(i)) {
				max = this.get(i);
			}
		}
	}
}
```

- From algorithms to collections' methods:
  - The cost of the static *getMax()* depends on the input, more exactly, it depends on the count of elements in the passed int[]! Clear!
  - To drive this point home for collections: let's assume *getMax()* is a fictive method of the type *List*.
  - We can easily spot only one difference: *getMax()* now operates on this, and no longer on a passed int[].
  - The costs of most algorithms depend on the count of passed elements!
  - The costs of most collection methods depend on the count of contained elements!

- In future lectures we'll discuss algorithms that are methods/member functions of collection types.
  - E.g. methods like *get()*/*set()*, *contains()*, *insert()*, *append()*, *remove()*

Thank you!