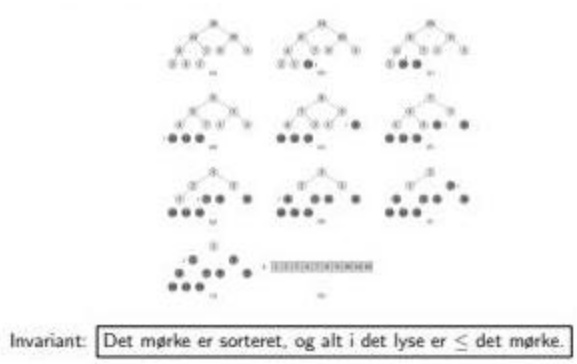
Invariants

An **invariant** is a condition that is maintained by an algorithm throughout (parts of) its execution. It often forms the core idea behind the algorithm.

Here are some examples of invariants across different algorithms:

- **Insertionsort**: The invariant is that everything to the left of the "black field" (a conceptual marker) is sorted. When the loop stops, the entire array is to the left of this field, implying the whole array is sorted.
- **Partition from Quicksort**: The invariant is that the light gray part is $\leq x <$ the dark gray part. After the loop, only x is white, with the rest being either light or dark gray. This divides the array into three parts: "$\leq x$", "$> x$", and x itself.
- **Build-Heap**: Subtrees whose root index is greater than the dark node comply with heap order. When the loop stops, the root of the entire tree has an index greater than the dark node, meaning the whole tree adheres to heap order.
- **Heapsort (and any Selection sort-based sorting)**:



  As shown in the image, the invariant is that the dark area is sorted, and everything in the light area is $\leq$ the dark area. When the loop stops, the entire array is dark, meaning it is sorted.

- **Searching in Binary Search Trees**: If the searched element k exists, it is in the subtree we have reached. The algorithm stops because it examines smaller and smaller subtrees. When it stops, either k is found, or an empty subtree is reached. If the latter, k does not exist in the tree.
- **Invariant during rebalancing after insertion in a Red-Black Tree**: There may be two red nodes in a row on a path somewhere in the tree. After k iterations, there are k fewer blacks between the problem and the root than at the beginning.
- **Invariant during rebalancing after deletion in a Red-Black Tree**: There may be one blackened node somewhere in the tree, and if the blackening is counted, the red-black requirements are met. After k iterations, there are k fewer blacks between the problem and the root than at the start.

The invariant here demonstrates several things that together ensure the algorithm's correctness:

- The same case analysis works every time, covering all possibilities so that the algorithm doesn't stall as long as the problem is not solved.
- The algorithm must stop, either because the problem has disappeared or because it has reached the root (where it is easily solved).

More formally, an invariant for an algorithm can be defined as:

> A statement about the contents of memory (variables, arrays, etc.) that is true after every step. At the algorithm's conclusion, correctness can be deduced from the statement (as well as the circumstances that caused the algorithm to stop).

Induction

To show that an invariant holds after every step, **induction** is used:

1. Invariant is upheld in the beginning.
2. Invariant is upheld before a step $\implies$ invariant is upheld after the step $\implies$ invariant is always upheld (where "step" is often an iteration of a loop).

Thus, to prove an invariant, show 1) and 2).

Induction is similar to the "Domino Principle":

1. Domino 1 falls.
2. Domino k falls $\implies$ domino $k+1$ falls.

Using Invariants

Invariants can be used at two different levels of detail (with a sliding transition between them):

1. As a tool to develop algorithm ideas: With the right invariant, the essence of the method is captured, and the algorithm merely needs to be written based on maintaining this invariant.
2. As a tool to write code (or detailed pseudo-code) and show that this specific code is correct.

At level 1, softer descriptions (text, figures) are appropriate. The examples given earlier illustrate level 1.

At level 2, one must write down the invariant precisely in terms of concrete variables from the code and argue via the concrete code's changes to these variables.

Example

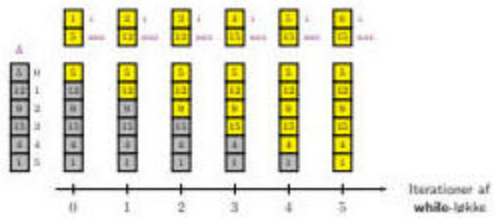
Let's illustrate level 2 with a simple example: finding the largest element in an array.

```
max = A[0]
i = 1
while i < A.length
    max = maximum(max, A[i])
    i++
```

Invariant:

> After the k'th iteration of the `while` loop, `max` contains the largest value of `A[0..(i - 1)]`.

This can be shown by induction on k.



The image shows how the maximum value is found iteratively.