Informatics II

Tutorial Session 6

Wednesday, 30th of March 2022

Discussion of Exercises 4 and 5,
Divide and Conquer, Heaps, Heapsort, Quicksort

14.00 – 15.45

BIN 0.B.06

Agenda

- Review of Exercise 4 (continued)
  - Master Method
  - Substitution Method
  - Divide and Conquer Algorithms
- Review of Exercise 5
- (Preview on Exercise 6)

Recurrences

- Master method recapitulation
- Master method recipe and example solved
- Substitution method / inductive proof example solved

Methods for Solving Recurrences

- Repeated substitution and looking sharply (also called iteration method / iterating the recurrence)
- Recursion tree – basically the same as repeated substitution, just graphically visualized
- Good guessing, followed by formal proof (also called «substitution method»); the proof can also be done as a follow-up when using the previous two methods (note that the terms «substitution method» and «repeated substitution» refer to two different methods although the sound quite similar)
- Master method: if it is a divide-and-conquer problem and you are only interested in order of growth (not an actual solution of the recurrence as defined before).
- (Characteristic equation: Certain types of recurrences – second-order linear homogeneous recurrence relations with constant coefficients – can be considered as a kind of polynomial and then a solution can be derived mechanically.)

Master Method: General Remarks

- The term «master method» (or even «master theorem») is rather exaggerative and pretentious; actually it's merely a kind of cooking recipe – and can be applied as such.
- It is a method specific for solving divide-and-conquer recurrences – and only them.
- Will yield asymptotic bound solution only and not an exact running time analysis as in the case of repeated substitution and recursion tree method.

Master Method: Interpretation

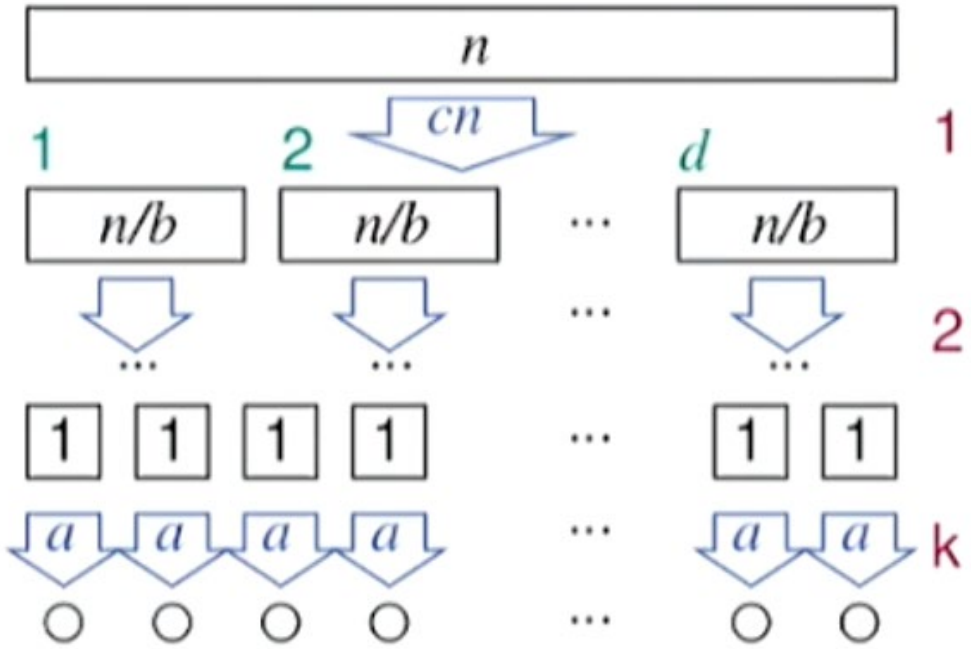
$$T(n) = a \cdot T(n/b) + f(n)$$

- n size of problem
- a number of subproblems
- b factor by which the problem size is reduced in recursive calls
- f(n) work done outside of recursive calls, work needed to split and merge

«The original problem of size n is split up into a new subproblems which are smaller by a factor of b. For splitting the original problem and merging of the solutions to the subproblems, work is needed which is described by f(n).»

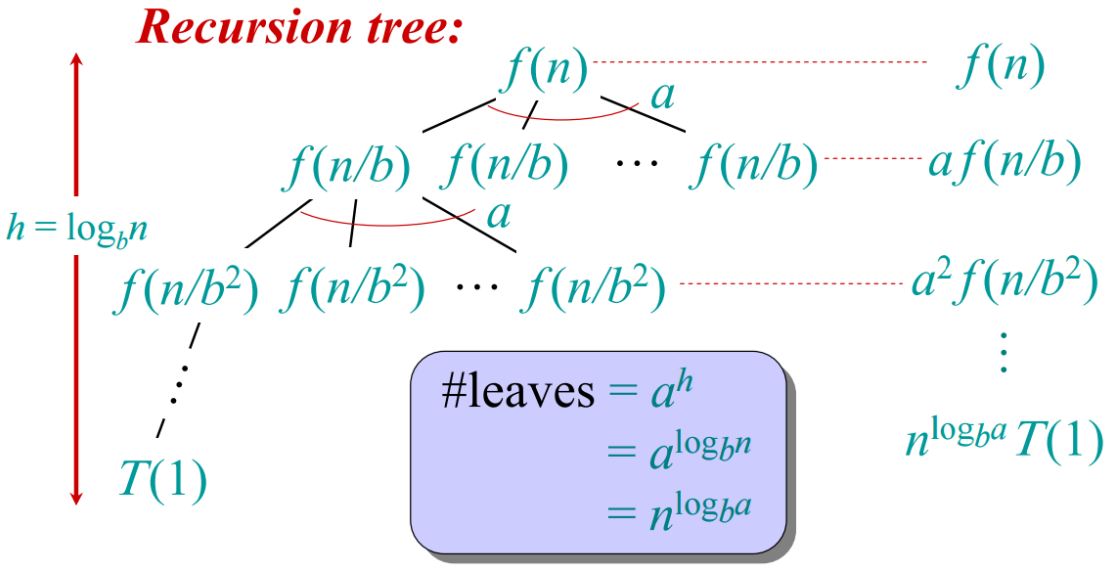
Master Method: Interpretation

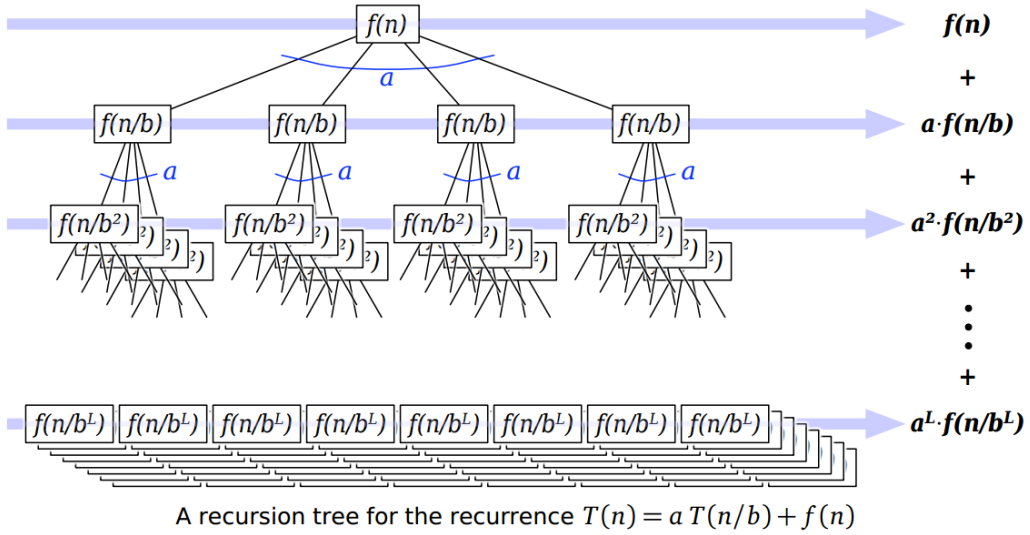
$$T(n) = a \cdot T(n/b) + f(n)$$



Master Method: Recursion Tree

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





A recursion tree for the recurrence $T(n) = a\,T(n/b) + f(n)$

Master Method: Prerequisites

Prerequisites for the application of the master method:

- a, b constants

  method not suited for variable resizing of problem

- a ≥ 1

  otherwise the original problem would be deconstructed into less than one subproblem

- b > 1

  otherwise subproblems will be bigger than the original problem

- f(n) asymptotically positive (i.e. for large enough n, f(n) is positive)

  negative work doesn't make sense

Master Method: Admissibility Examples

Decide whether the master method can be applied to the following recurrences:

- $T_1(n) = 4n \cdot T_1(n/3) + n^{2n}$
  → not applicable: a = 4n is not a constant
- $T_2(n) = ½ \cdot T_2(n/3) + n/2$
  → not applicable: a < 1
- $T_3(n) = 3 \cdot T_3(n/2^{-1}) + \log(n)$
  → not applicable: b < 1
- $T_4(n) = 3 \cdot T_4(n - 3) + n$
  → not applicable: wrong format / parsing fails
- $T_5(n) = 4 \cdot T_5(n/4) - n^2$
  → not applicable: f(n) = $-n^2$ is not asymptotically positive
- $T_6(n) = 5 \cdot 2^n + 5 \cdot T_6(n/5) + n^2$
  → applicable: a = 5, b = 5, f(n) = $5 \cdot 2^n + n^2$
- $T_7(n) = T_7(n/2) + 42$
  → applicable: a = 1, b = 2, f(n) = 42

Master Method: Three Cases

First, calculate $x = \log_b(a)$. Then compare the non-recursive work $f(n)$ to $n^x = n^{\log_b(a)}$.

(Note that n^x is the number of leaves in the corresponding recursion tree.)

- **Case ①**: $f(n)$ grows polynomially slower than n^x.

  formally: $\exists \varepsilon > 0: f(n) \in O(n^{x-\varepsilon})$

  → **$T(n) \in \Theta(n^x)$**

- **Case ②**: $f(n)$ grows (about) as fast as n^x.

  formally: $\exists \varepsilon > 0: f(n) \in \Theta(n^{x-\varepsilon} \cdot (\log(n))^k)$ for some $k \geq 0$

  → **$T(n) \in \Theta(n^x \cdot (\log(n))^{k+1})$** very often: $k = 0$

- **Case ③**: $f(n)$ grows polynomially faster than n^x.

  formally: $\exists \varepsilon > 0: f(n) \in \Omega(n^{x-\varepsilon})$

  → **$T(n) \in \Theta(f(n))$**

Master Method: Three Cases

Remarks:

- Case ③ will require an additional regularity check.
- The three cases correspond to different distributions of work in the recursion tree:
  - Case ①: cost at the leaves dominating
  - Case ②: cost evenly distributed
  - Case ③: cost at the root dominating
- Note that it is not sufficient if f(n) just grows somehow slower or faster than n^x. It needs to be *polynomially* slower or faster.

Master Method: Cooking Recipe

1) Look, think, interpret
   To what kind of problem solving approach does the given recurrence relation correspond to (if any); is it a divide-and-conquer problem?
2) Pattern matching
   Try to find the a, b and f(n) parts in the given recurrence relation.
3) Check parameters
   Are the found parameters a, b, f(n) eligible for the application of the master method?
4) Determine the case
   Compare asymptotically n^x where $x = \log_b(a)$ with f(n).
   a) Perform additional regularity check if it is case ③
5) Write down the solution

Master Method: Application Example

Given: $T(n) = 7 \cdot T(n/2) + n^2$ (e.g.: Strassen's algorithm for matrix multiplication)

- *Step 1: Look and think:*
  «The Problem of size n is split up into 7 subproblems, each of which has half the size of the original problem; the work for splitting and merging is quadratic with regard to the problem size.» → seems ok on first look

- *Step 2: Pattern matching:*
  $a = 7$, $b = 2$, $f(n) = n^2$ → successful

- *Step 3: Check parameters:*
  $a \geq 1$, const ✓; $b > 1$, const ✓; $f(n)$ asymptotically positive ✓ → successful

- *Step 4: Determine the case:*
  $x = \log_b(a) = \log_2(7) \approx 2.807$
  Compare: $f(n) = n^2$ to $g(n) = n^x \approx n^{2.807}$. ($n^x$ is the number of leaves in the recursion tree)
  Obviously, $f(n)$ grows polynomially slower than n^x.
  $\exists \varepsilon > 0: f(n) \in O(n^{x-\varepsilon})$ → e.g. $\varepsilon = 0.8$
  ⇒ Case ①, running time dominated by the cost at the leaves

- *Step 5: Write down result:*
  $T(n) \in \Theta(n^x) = \Theta(n^{\mathrm{ld}(7)}) \approx \Theta(n^{2.807})$

Master Method: Technicalities

- The base case oftentimes is not described when dealing with the master method. (You could argue that all respective recurrences are actually incomplete.)

  → It is just assumed that T(0) will only affect the result by some constant which can be ignored asymptotically.

- Considerations on rounding of parameters and consequently floor and ceiling functions (Gauß-Klammern) are usually omitted / ignored.

  → It is just written T(n/b) instead of T(⌊n/b⌋) or T(⌈n/b⌉).

Exercise 4, Task 5: Master Method

(a) $T(n) = 32T(n/4) + \sqrt{n^5} + 2n + 7$

→ applicable: a = 32, b = 4, f(n) = $n^{2.5}$ + 2n + 7
case 2, T(n) ∈ Θ($n^{2.5}$ · log(n))

(b) $T(n) = 2T(n/3) \cdot \log n + 5n^4 + 1$

→ not applicable: a is not a constant

(c) $T(n) = 36T(n/6) + 2\sqrt{n}\log(n)$

→ applicable: a = 36, b = 6, f(n) = 2 · sqrt(n) · log(n)
case 1, T(n) ∈ Θ(n^2)

(d) $T(n) = 2T(5n/4) + n$

→ not applicable: b < 1

(e) $T(n) = \frac{\sqrt{3}}{2}T(n/7) + 2^n$

→ not applicable: a < 1

(f) $T(n) = 21n + 7T(3n/11) + 8\log(n!) + \sqrt{\log(n)} + n^2$

→ applicable: a = 7, b = 11/3, f(n) = 21n + 8log(n!) + sqrt(log(n)) + n^2
case 3, T(n) ∈ Θ(n^2)

Substitution Method / Inductive Proof: Example

Task 1.4b from midterm 1 of FS 2016:

Consider the following recurrence: $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/5) + T(7n/10) + n & \text{if } n > 1 \end{cases}$

Prove the correctness of the estimate O(n) using induction (i.e. show that T(n) is in O(n)).

Note that we're given two things: a recurrence relation and an estimate / guess for the upper bound of T(n).

We will proceed in three steps:

1 Apply definition of asymptotic complexity and fill in the guess which is to be proofen

→

2 Assume, the guess was true (inductive step) and calculate the consequences of this

→

3 Find a constant c which fulfills the assumption from (2) for any n which is big enough

Substitution Method / Inductive Proof: Example

Consider the definition of big O notation:

$T(n) \in O(g(n))$ iff $\exists$ constants $n_0 > 0$, $c > 0$ such that $T(n) \leq c \cdot g(n) \quad \forall n \geq n_0$

Our goal is to proof that the recurrence $T(n)$ is in $O(g(n)) = O(n)$, i.e. that $T(n)$ is bounded from above by a linear function $g(n) = n$.

This is true if, for any sufficiently big n, we can find a constant c such that $T(n) \leq c \cdot g(n) = c \cdot n$.

↑
guess which is to proof by finding a suitable witness constant c

$T(n) \in O(g(n))$ iff $\exists$ constants $n_0 > 0$, $c > 0$ such that $T(n) \leq c \cdot g(n) \quad \forall n \geq n_0$

Substitution Method / Inductive Proof: Example

Let's assume that the assumption holds that T(n) can be bounded from above by g(n) = n.

If this is the case, we can always replace any occurrence of the term T(n) by g(n) = c·n and be sure that the result will be smaller or equal to what was written there before (by definition of big O).

$$T(n) \leq c \cdot g(n)$$

(This step constitutes the inductive step of a proof by induction.)

When this is applied to the given recurrence, this yields the following:

$$T(n) = T(n/5) + T(7n/10) + n \leq c \cdot g(n/5) + c \cdot g(7n/10) + n = c \cdot n/5 + c \cdot 7n/10 + n =$$
$$= c \cdot 9n/10 + n =$$
$$= n \cdot (0.9 \cdot c + 1)$$

Replace T(n/k) with n/k
(Note that if T(n/k) is to be replaced, this has to be replaced by n/k.)

Replace g(n) with the guess and include the constant c

Substitution Method / Inductive Proof: Example

From inductive step, we know want to find a witness constant c for which

$$(0.9 \cdot c + 1) \cdot n \leq c \cdot n$$

Let's just try with c = 10 and fill this into the inequality from above:

$$\begin{aligned}(0.9 \cdot 10 + 1) \cdot n &\leq 10 \cdot n \\ (9 + 1) \cdot n &\leq 10 \cdot n \\ 10 \cdot n &\leq 10 \cdot n\end{aligned}$$

...which is obviously true for any $n \geq 0$ (i.e. the threshold $n_0 > 0$ can be chosen arbitrarily here) and thus the required proof has succeded.

For any choice of c which is bigger than 10, the inequality holds even clearer, of course.

Therefore, we have found a withness c which shows that $g(n) = c \cdot n = 10 \cdot n$ bounds from above the recurrence T(n).

Substitution Method / Inductive Proof: Example

But how come, we've chosen c = 10 here? How can How can we systematically find such a constant c?

The constant c can be found systematically by getting rid of the variable n in the inequality and solving for the constant c. Consider the following sequence of reformattings:

$$
\begin{array}{lcll}
(0.9 \cdot c + 1) \cdot n & \leq & c \cdot n & | : n \\
(0.9 \cdot c + 1) & \leq & c & | - 0.9 \cdot c \\
1 & \leq & c - 0.9 \cdot c & | \text{ subtraction} \\
1 & \leq & 0.1 \cdot c & | \cdot 10 \\
10 & \leq & c &
\end{array}
$$

Exercise 4, Task 2a: Substitution Method

$$
\begin{aligned}
&\textit{To show:} && T(n) \in O(g(n)) \text{ where } g(n) = n^2 \\
&\textit{Hypothesis:} && T(n) \leq c \cdot g(n) = c \cdot n^2 \text{ for some constant } c > 0 \\
&\textit{Inductive step:} && T(n) = 3T(2n/3) + 5n^2 + 4n \\
& && \leq 3 \cdot c \cdot g(2n/3) + 5n^2 + 4n \\
& && = 3 \cdot c \cdot (2n/3)^2 + 5n^2 + 4n \\
& && = (\tfrac{4c}{3} + 5)n^2 + 4n
\end{aligned}
$$

Contradict the hypothesis: use $\tilde{g}(n) = c \cdot n^2 - d \cdot n$ instead of $g(n)$:

$$T(n) \leq (c \cdot \tfrac{4}{3} + 5)n^2 - d \cdot \tfrac{2}{3}n + 4n \not\leq c \cdot n^2 - d$$

There is no suitable witness constant c such that $T(n) \leq c \cdot n^2$ holds, therefore $T(n) \notin O(n^2)$.

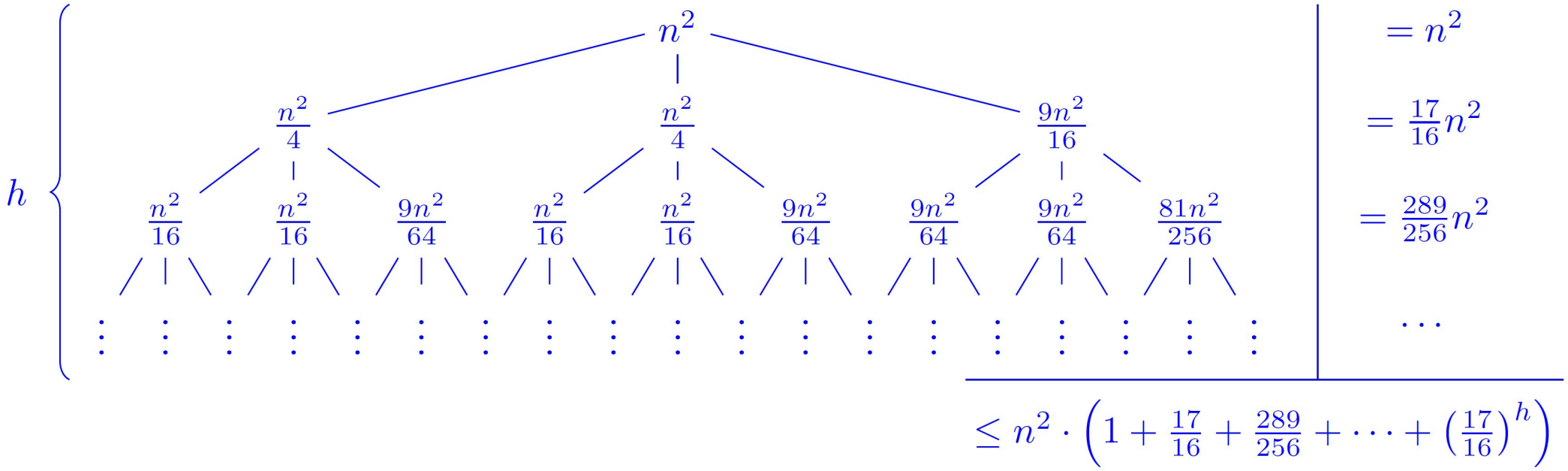
Exercise 4, Task 2b: Substitution Method

$$
\begin{aligned}
&\textit{To show:} && T(n) \in O(g(n)) \text{ where } g(n) = \log(n) \\
&\textit{Hypothesis:} && T(n) \leq c \cdot g(n) = c \cdot \log(n) \text{ for some constant } c > 0 \\
&\textit{Inductive step:} && T(n) = T(n/2) + 3T(3n/7) + 2n \\
& && \leq c \cdot g(n/2) + 3 \cdot c \cdot g(3n/7) + 2n \\
& && = c \cdot \log(n/2) + 3 \cdot c \cdot \log(3n/7) + 2n \\
& && \not\leq c \cdot \log(n)
\end{aligned}
$$

Since c cannot be chosen such that $2n \leq c \cdot \log(n)$ holds for growing values of n, it is concluded that $\mathbf{T(n) \notin O(\log(n))}$.

Exercise 4, Task 4a: Recursion Tree Method

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + T(3n/4) + n^2 & \text{if } n > 1 \end{cases}$$



Exercise 4, Task 4a: Recursion Tree Method

The sum $\left(1 + \frac{17}{16} + \frac{289}{256} + \cdots + \left(\frac{17}{16}\right)^h\right) = \sum_{i=0}^{h} \left(\frac{17}{16}\right)^i = \sum_{i=0}^{h} q^i$ is a finite geometric series. Since $|q| = \frac{17}{16} > 1$, the respective infinite geometric series $\sum_{i=0}^{\infty} q^i$ diverges and thus cannot be used to find an upper bound. The finite geometric series $\sum_{i=0}^{h} q^i$ is known to sum up to $\frac{q^{h+1}-1}{q-1}$. The recursion tree has its biggest height / longest branch on the left-hand side where it grows until $\frac{n^2}{4^h} = 1$ which implies $h = 2\log_4(n)$. To estimate the total work in the recursion tree's nodes, we will assume the tree has the height of the leftmost branch everywhere and this overestimation will yield an upper bound for the total work in the tree's nodes:

$$
\begin{aligned}
T(n) &< n^2 \cdot \sum_{i=0}^{h} \left(\frac{17}{16}\right)^i = n^2 \frac{\left(\frac{17}{16}\right)^{h+1} - 1}{\frac{17}{16} - 1} \\
&= n^2 \frac{\left(\frac{17}{16}\right)^{2\log_4(n)} - 1}{\frac{1}{16}} = 16n^2 \cdot \left(n^{2\log_4\left(\frac{17}{16}\right)} - 1\right) \\
&\approx 16n^{4.044} - 16n^2 \in O(n^5)
\end{aligned}
$$

Exercise 4, Task 4b: Proof for Result from Recursion Tree Method

To show: $T(n) \in O(g(n))$ where $g(n) = n^5$

Hypothesis: $T(n) \leq c \cdot g(n) = c \cdot n^5$ for some constant $c > 0$

Inductive step:

$$\begin{aligned} T(n) &= 2T(n/2) + T(3n/4) + n^2 \\ &\leq 2 \cdot c \cdot g(n/2) + c \cdot g(3n/4) + n^2 \\ &= 2 \cdot c \cdot (n/2)^5 + c \cdot (3n/4)^5 + n^2 \\ &= \tfrac{307c}{1024} n^5 + n^2 \\ &\not\leq c \cdot n^5 \end{aligned}$$

Strengthen the hypothesis: Instead of $g(n) = n^5$ use $\tilde{g}(n) = n^5 - d \cdot n^2$ with $d > 0$ and show that $T(n) \in O(\tilde{g}(n))$ which implies $T(n) \in O(g(n))$:

$$\begin{aligned} T(n) &= 2T(n/2) + T(3n/4) + n^2 \\ &\leq c \cdot 2 \cdot \tilde{g}(n/2) + c \cdot \tilde{g}(3n/4) + n^2 \\ &= \tfrac{307c}{1024} n^5 - \tfrac{17d}{16} n^2 + n^2 = \tfrac{307c}{1024} n^5 - \tfrac{d}{16} n^2 \\ &\leq c \cdot n^5 - d \cdot n^2 \text{ for } d \geq 1,\ c \geq 1,\ n \geq 1 \end{aligned}$$

Hence: $\mathbf{T(n) \in O(n^5)}$

Recurrences / Master Method: Additional Practice

Additional examples for practice can be found here:

https://h5p.org/node/457330

Divide and Conquer

- Basic Structure of Divide and Conquer Algorithms
- Example of a Divide and Conquer Algorithm
- Task 6: Sorting Coloured Circles
- Task 7: Modified Merge Sort Algorithm

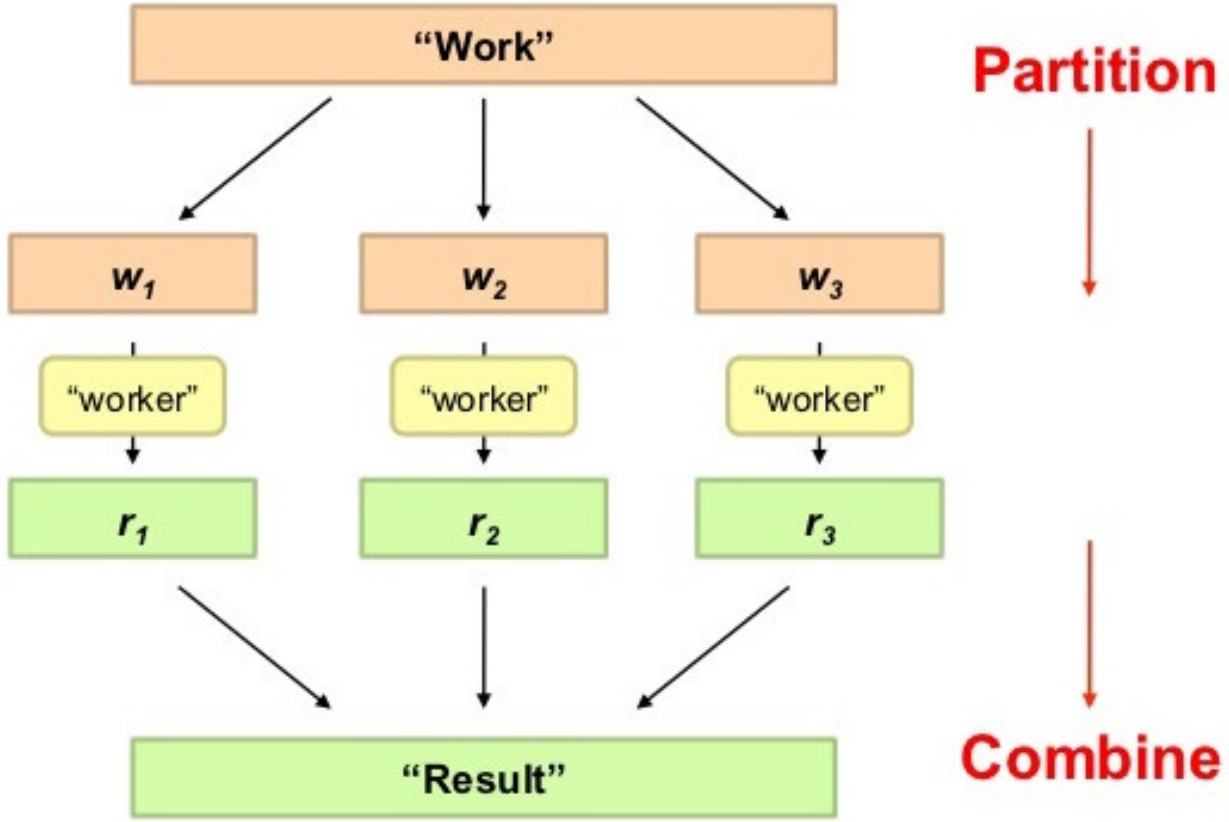
Divide and Conquer Algorithms

Divide and conquer algorithms consist of three (or four) parts:

- **(1) divide:** split up the problem into subproblems (e.g. in the middle of an array)
- **(2) conquer:** *recursively* apply the solution to the subproblems (until the base case is reached)
- **(3) combine:** merge the solutions from smaller subproblems into the solution of a bigger subproblem

- **(2a) base case:** abort recursion when a minimal problem is reached which can be solved directly

Divide and Conquer Algorithms



Template for Divide and Conquer Algorithms

Algorithm: *divcon*(problem, start, end)

```
if is_small(problem, start, end) then
    return solution(problem, start, end)
else
    middle = divide(problem, start, end)
    x = divcon(problem, start, middle)
    y = divcon(problem, middle + 1, end)
    result = combine(x, y)
    return result
```

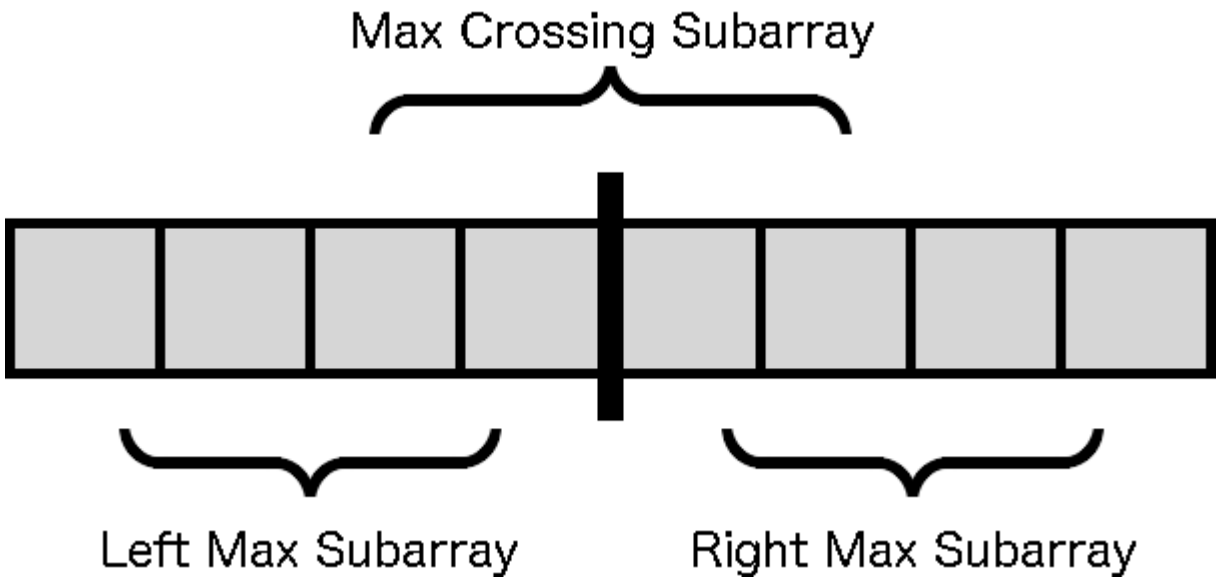
base case

divide

conquer

combine

Divide and Conquer Algorithms: Example



Exercise 4, Task 1a: What is a Divide and Conquer Algorithm?

Is the following statement true or false?

“Divide and conquer technique means that an algorithm is divided into subroutines (functions) which solve similar tasks with the goal that the algorithm is easier understood and easier to debug.”

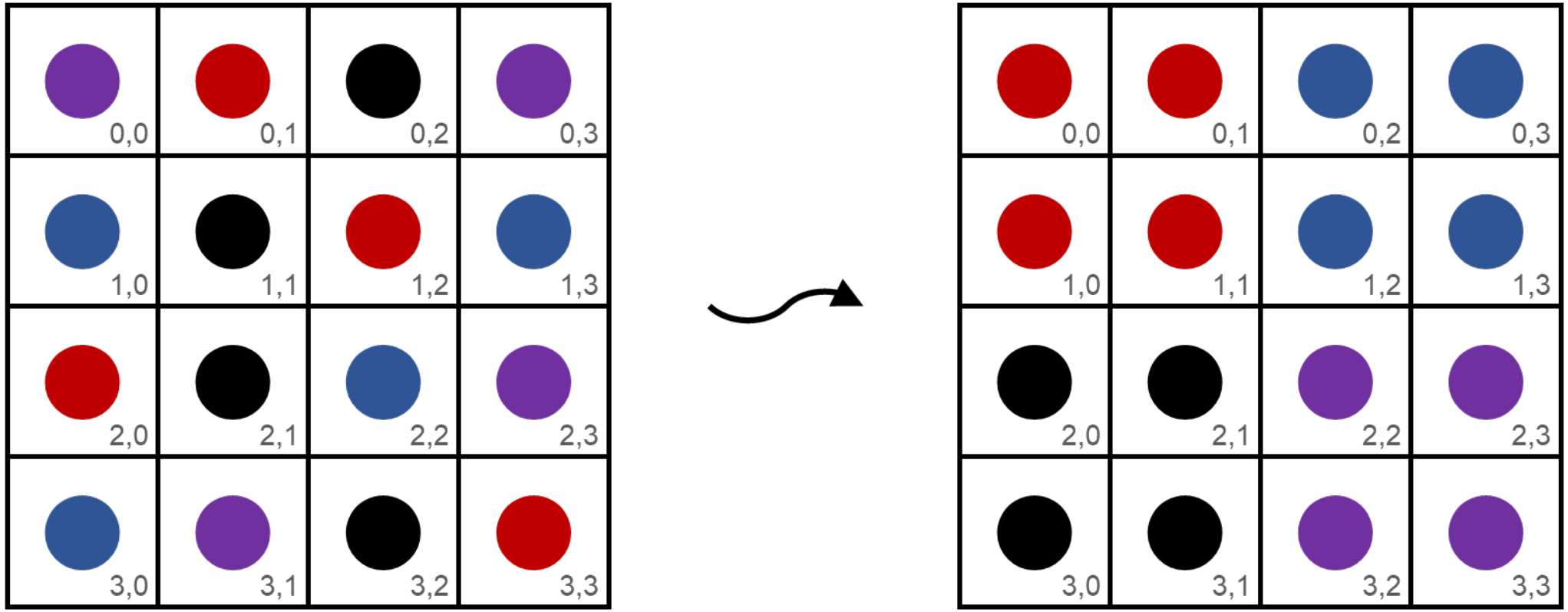
→ false

Exercise 4, Task 1b: Recursion Depth of Merge Sort

Consider an arbitrary array A of n = 543 integers which is sorted in ascending order using the merge sort algorithm as outlined in the lecture. What will be the maximum recursion depth reached during the algorithm?

→ $d(n) = \lceil \log_2(n) \rceil = \lceil \log_2(543) \rceil \approx \lceil 9.085 \rceil =$ **10**

Exercise 4, Task 6: Sorting Coloured Circles



Exercise 4, Task 6: Sorting Coloured Circles

Algorithm: segregate($arr[0..m-1, 0..m-1], row\_from, row\_to, col\_from, col\_to$)

```
1  current_size = row_to − row_from + 1
2  if current_size == 2 then
3      if arr[row_from, col_from + 1] == red then
4          exchange(arr[row_from, col_from], arr[row_from, col_from + 1])
5      else if arr[row_from + 1, col_from] == red then
6          exchange(arr[row_from, col_from], arr[row_from + 1, col_from])
7      else if arr[row_from + 1, col_from + 1] == red then
8          exchange(arr[row_from, col_from], arr[row_from + 1, col_from + 1])
9      if arr[row_from + 1, col_from] == blue then
10         exchange(arr[row_from, col_from + 1], arr[row_from + 1, col_from])
11     else if arr[row_from + 1, col_from + 1] == blue then
12         exchange(arr[row_from, col_from + 1], arr[row_from + 1, col_from + 1])
13     if arr[row_from + 1, col_from + 1] == black then
14         exchange(arr[row_from + 1, col_from], arr[row_from + 1, col_from + 1])
15     return
16 row_middle = row_from + current_size/2 − 1
17 col_middle = col_from + current_size/2 − 1
18 segregate(arr, row_from, row_middle, col_from, col_middle)
19 segregate(arr, row_from, row_middle, col_middle + 1, col_to)
20 segregate(arr, row_middle + 1, row_to, col_from, col_middle)
21 segregate(arr, row_middle + 1, row_to, col_middle + 1, col_to)
22 for i = row_from to row_to do
23     for j = col_from + current_size/4 to col_from + current_size/2 − 1 do
24         exchange(arr[i, j], arr[i, j + current_size/4])
25 for i = col_from to col_to do
26     for j = row_from + current_size/4 to row_from + current_size/2 − 1 do
27         exchange(arr[i, j], arr[i + current_size/4, j])
```

base case

divide

conquer

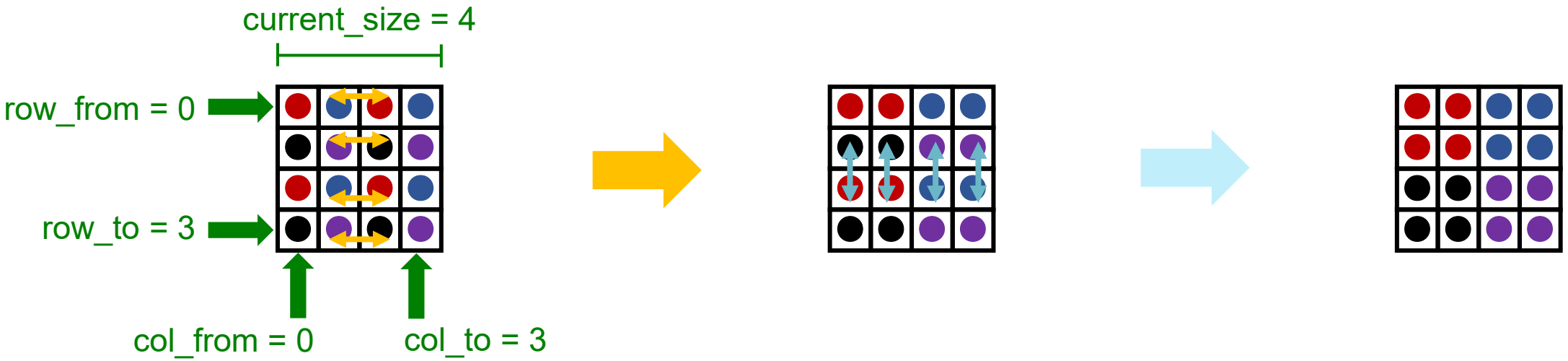
combine

```
22 for i = row_from to row_to do
23     for j = col_from + current_size/4 to col_from + current_size/2 − 1 do
24         exchange(arr[i, j], arr[i, j + current_size/4])
```

```
25 for i = col_from to col_to do
26     for j = row_from + current_size/4 to row_from + current_size/2 − 1 do
27         exchange(arr[i, j], arr[i + current_size/4, j])
```

for all rows, switch the two columns in the middle

for all columns, switch the two rows in the middle

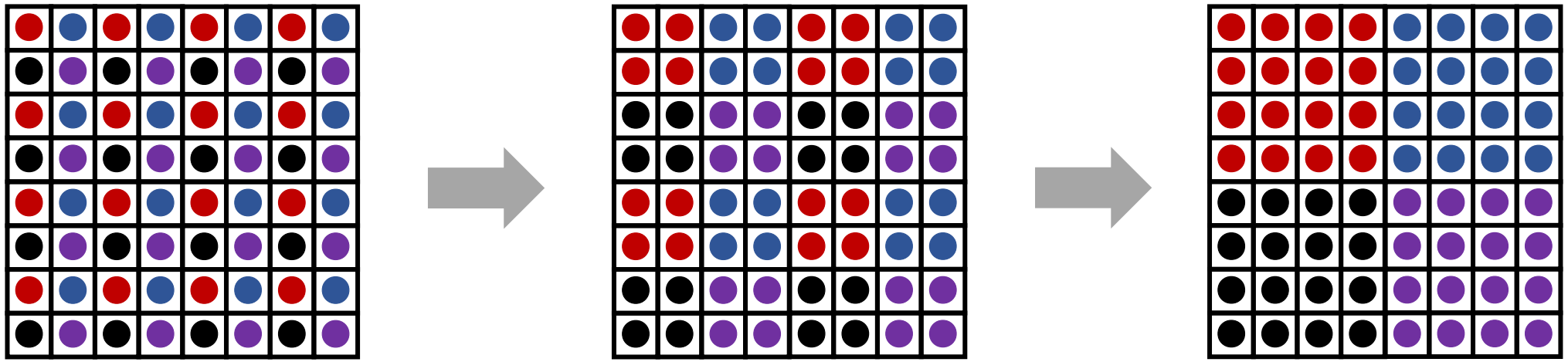


Remark: This also works the other way around.

Exercise 4, Task 6: Sorting Coloured Circles: Recursion Tree

```
18 segregate(arr, row_from, row_middle, col_from, col_middle)
19 segregate(arr, row_from, row_middle, col_middle + 1, col_to)
20 segregate(arr, row_middle + 1, row_to, col_from, col_middle)
21 segregate(arr, row_middle + 1, row_to, col_middle + 1, col_to)
```

Exercise 4, Task 6: Sorting Coloured Circles



Exercise 4, Task 6: Sorting Coloured Circles

Additional exercises:

a) What is the recurrence which models the divide and conquer algorithm for solving this task?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 4 \\ 4T(n/4) + d \cdot n + \Theta(1) & \text{if } n > 4 \text{ and with a constant } d > 0 \end{cases}$$

b) What is the asymptotic complexity of this algorithm?

Exercise 4, Task 6: Sorting Coloured Circles

Additional exercise: What would be other approaches (than the divide and conquer technique) to solve this problem? Are they better, worse or comparable with regard to time complexity?

Example: randomly permutate all circles in the matrix and check whether it is (happens to be) sorted as required
-> Bestcase: O(n), if matrix happens to be correct after first permutation (n permutations and n checks)
-> Average case: O((n-1)*n!)
-> Worst case: does not terminate (algorithm does not fulfill total correctness property; the algorithm is almost certain to terminate at some point but in principle, with a very slim chance, this could never happen)

Exercise 4, Task 7a: Modified Merge Sort Algorithm

(a) Implement in C the merge sort algorithm as given in the lecture slides and modify the algorithm as follows:

(i) It shall be possible to choose the order of sorting (ascending or descending) through a parameter.

(ii) Any input array should first be checked whether it is already sorted in the required order using a linear search. If the former is the case, an early abort of the algorithm shall be performed.

(iii) Once the divide stage of the merge sort algorithm has reached a subproblem size (partial array length) smaller or equal to 6, the recursion should be aborted and the subproblem should be solved using the insertion sort algorithm instead.

Exercise 4, Task 7a: Modified Merge Sort Algorithm

Algorithm: merge_sort_mod($input[0..n-1], left, right, direction$)

```
1  threshold = 6
2  if is_sorted(input, left, right, direction) then
3      return
4  else if left < right then
5      if right − left ≤ threshold then
6          insertion_sort(input, left, right, direction)
7      else
8          middle = ⌊(left + right)/2⌋
9          merge_sort_mod(input, left, middle, direction, temp)
10         merge_sort_mod(input, middle + 1, right, direction, temp)
11         merge(input, left, right, middle, direction, temp)
```

Exercise 4, Task 7a: Modified Merge Sort Algorithm

```
int is_sorted(int input[], int left, int right, const int direction) {
      for (int i = left + 1; i <= right; i++) {
            if ((input[i - 1] > input[i] && direction == ASCENDING) ||
                  (input[i - 1] < input[i] && direction == DESCENDING)) {
                return 0;
            }
      }
      return 1;
}
```

Exercise 4, Task 7a: Modified Merge Sort Algorithm

```
int is_put_before(int a, int b, const int direction) {
	if ((direction == ASCENDING && a < b) || (direction == DESCENDING && a > b)) {
		return 1;
	}
	else {
		return 0;
	}
}
```

Exercise 4, Task 7a: Modified Merge Sort Algorithm

```
void insertion_sort(int input[], int left, int right, int direction) {
        for (int i = left; i <= right; i++) {
                int j = i - 1;
                int temp = input[i];
                while (j >= left && is_put_before(temp, input[j], direction)) {
                        input[j + 1] = input[j];
                        j--;
                }
                input[j + 1] = temp;
        }
}
```

Exercise 4, Task 7a: Modified Merge Sort Algorithm

```
void merge(int input[], int left, int right, int middle, int direction, int temp[]) {
        for (int i = left; i <= middle; i++) {
                temp[i] = input[i];
        }
        for (int i = middle + 1; i <= right; i++) {
                temp[right + middle + 1 - i] = input[i];
        }
        int i = left;
        int j = right;
        for (int k = left; k <= right; k++) {
                if (is_put_before(temp[i], temp[j], direction)) {
                        input[k] = temp[i];
                        i++;
                }
                else {
                        input[k] = temp[j];
                        j--;
                }
        }
}
```

Exercise 4, Task 7a: Modified Merge Sort Algorithm

```
void merge_sort_mod(int input[], int left, int right, int direction, int temp[]) {
        const int threshold = 6;
        if (is_sorted(input, left, right, direction)) {
                return;
        }
        else if (left < right) {
                if (right - left <= threshold) {
                        insertion_sort(input, left, right, direction);
                }
                else {
                        int middle = (left + right) / 2;
                        merge_sort_mod(input, left, middle, direction, temp);
                        merge_sort_mod(input, middle + 1, right, direction, temp);
                        merge(input, left, right, middle, direction, temp);
                }
        }
}
```

Exercise 4, Task 7b: Asymptotic Analysis

(b) Give the best case, average case and worst case time complexity of the algorithm implemented in the previous subtask with regard to the size of the input array.

- Best case: O(n)
- Average case: O(n log(n))
- Worst case: O(n log(n))

Exercise 4, Task 7: Modified Merge Sort Algorithm: Remarks

The algorithm implemented here is an example of a hybrid sorting algorihm. An algorithm which works quite simlarly is for example Timsort which is the algorithm which is used in Python for sorting.

Heaps

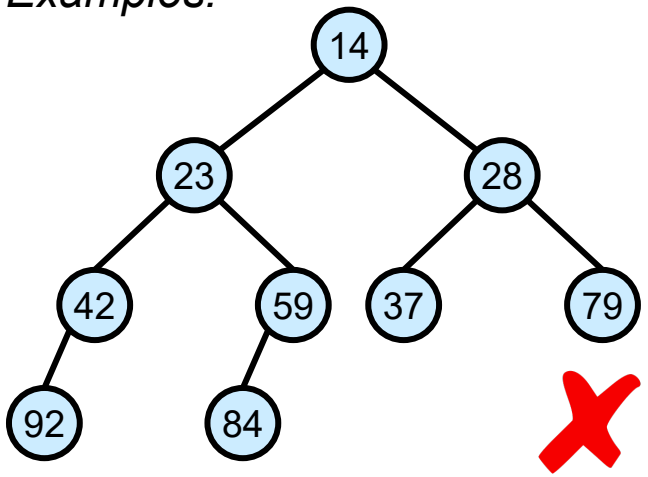
- Heap Conditions
- Representation of a Heap as an Array
- Heap Operations
- Heapsort

Heap Conditions

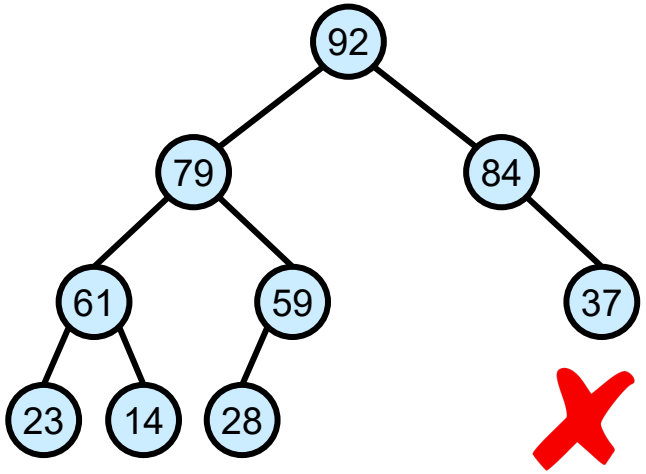
– A (binary) heap is a binary tree.

– All levels except the lowest are fully filled and the lowest level is occupied starting from left without gaps (= «nearly complete tree»).

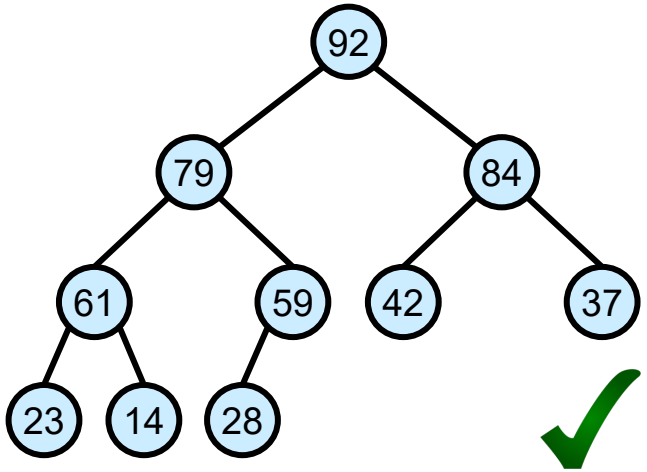
Examples:



→ not a binary heap



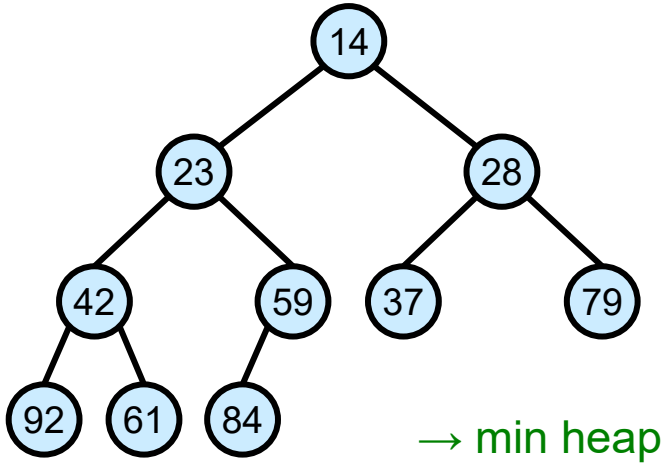
→ not a binary heap



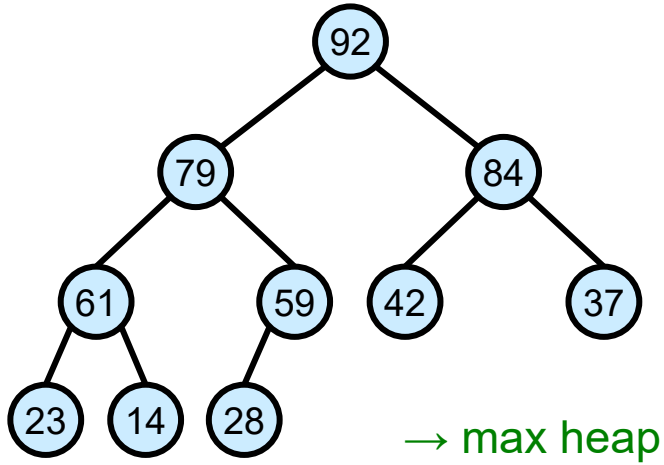
→ valid binary max heap

Heap Conditions

- The keys stored in the nodes of the heap are ordered:
  - in a max-heap: for all nodes, the key is bigger than or equal to all its children
    → largest element at root
  - in a min-heap: for all nodes, the father node is always smaller than or equal to his sons
    → smallest element at root

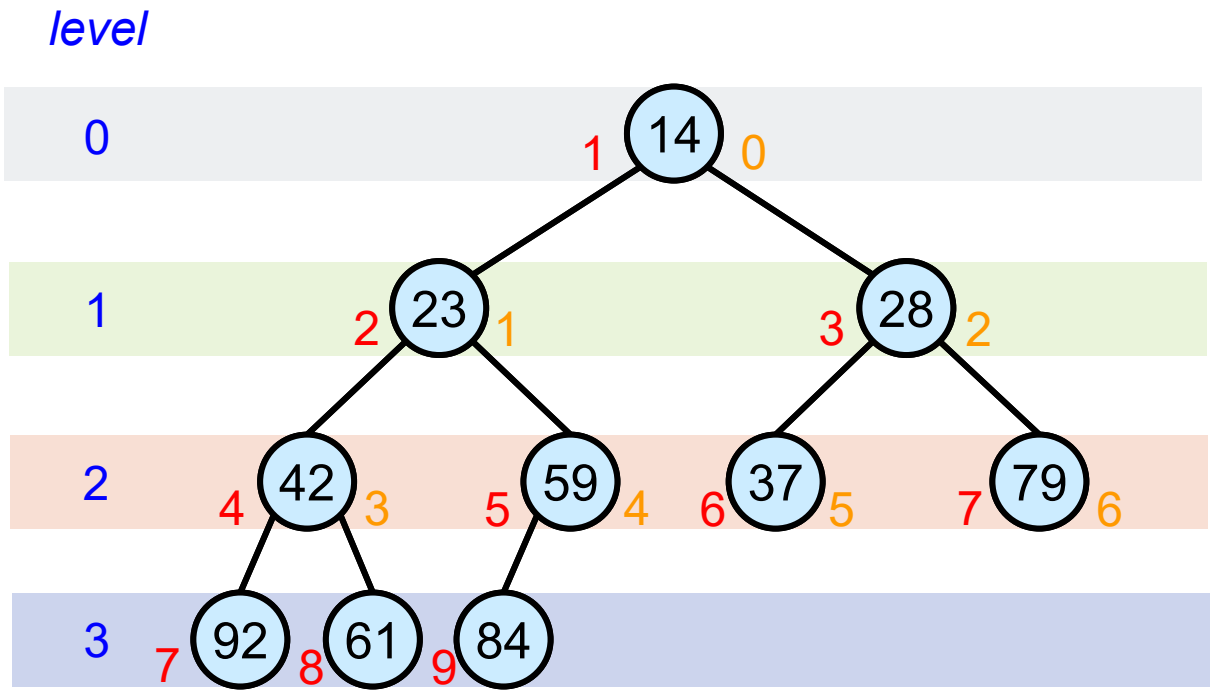


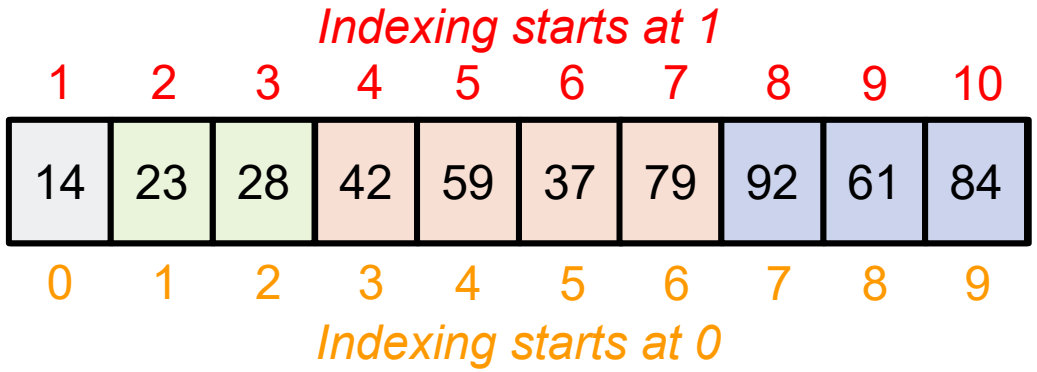
→ min heap



→ max heap

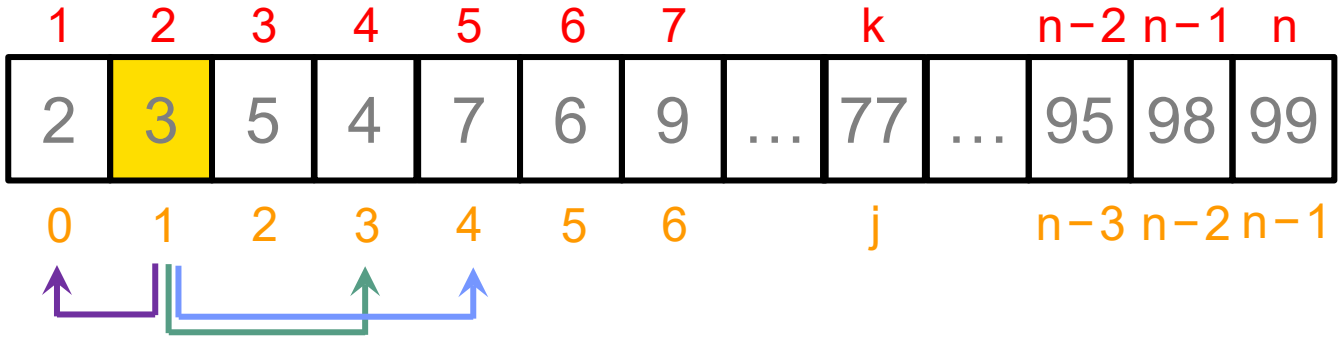
Representation of a Heap as an Array

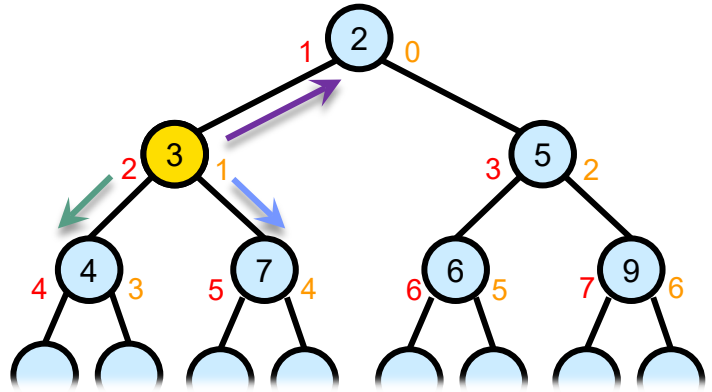




Note: This kind of tree traversal is called level order.

Addressing Parent and Child Nodes





	Indexing starts at 1	*Indexing starts at 0*
Getting to parent of node with index k	$\lfloor k / 2 \rfloor$	$\lfloor (j - 1) / 2 \rfloor$
Getting to left child of node with index k	$2 \cdot k$	$2 \cdot j + 1$
Getting to right child of node with index k	$2 \cdot k + 1$	$2 \cdot j + 2$

Heap Conditions: Exercise

The following arrays shall each represent a binary heap. Do they fulfill the conditions of a heap?

A:

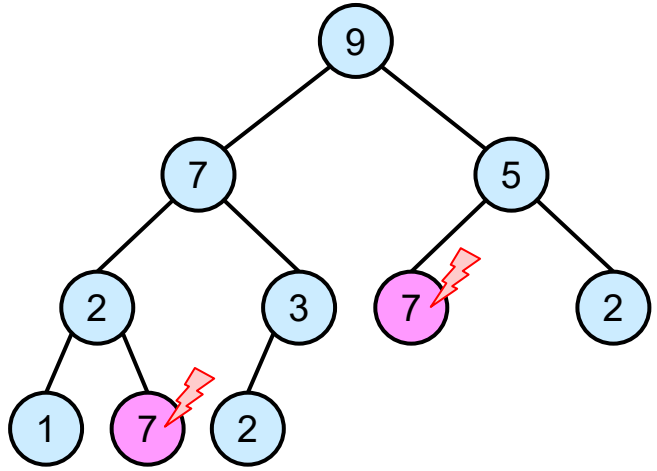
7	1	1	2	4	9	3	3	6	5

→ Not a heap, because the first element is neither the maximum nor the minimum of the elements

B:

9	7	5	2	3	7	2	1	7	2

→ By drawing the corresponding heap, we can see that elements at indices 5 and 8 (starting from zero) violate the heap condition. Therefore it is not a heap.



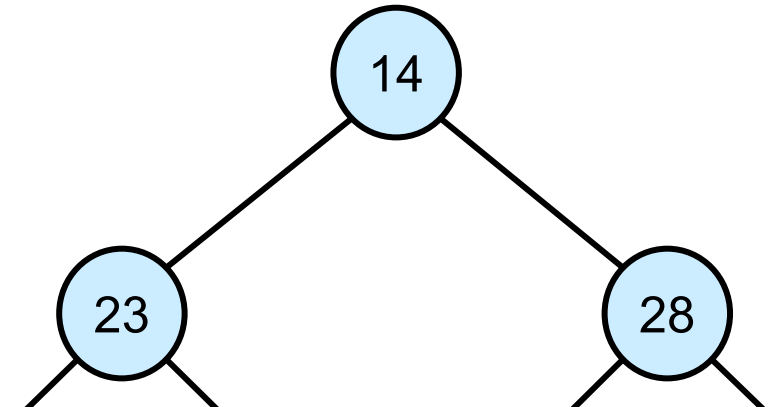
Heap Conditions: Comprehension Question

What is the minimum and the maximum number of nodes which a binary heap of height h (i.e. with h+1 levels) can have?

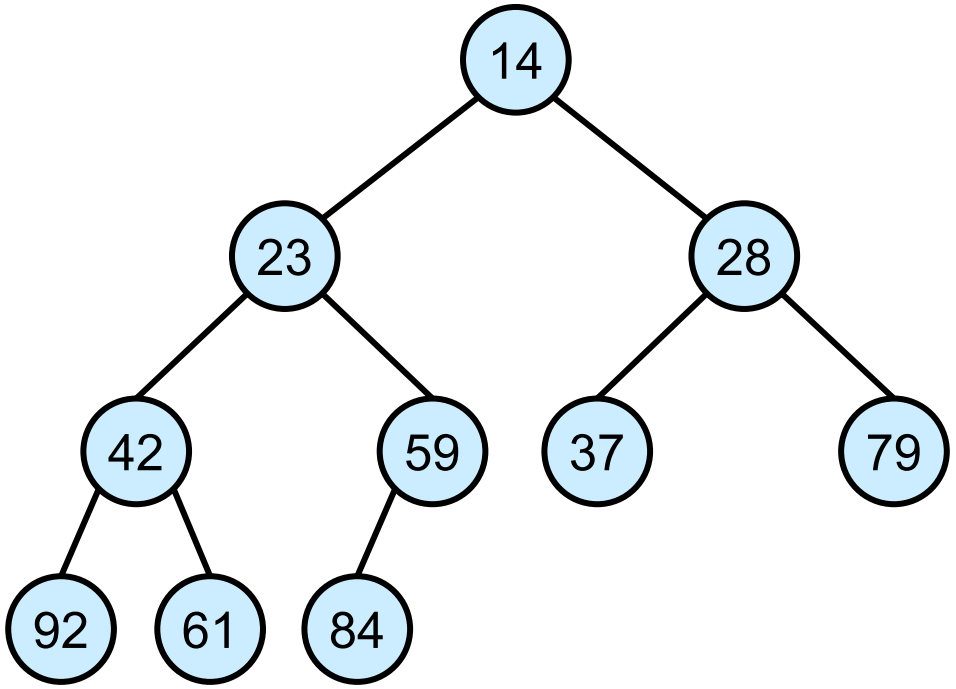
Heaps: Removing the Root

1) Remove the key from the root node.
2) Take the value in the last node and (temporarily) store it in the root node.
3) Let the value at the root node «trickle down» in the tree by swapping it with the value from the smaller (in case of min-heap) or bigger (max-heap) child node until both children are bigger (min-heap) or smaller (max-heap) than it. This process is also called to «heapify» the array / tree.

Remark: When implementing this as part of heap sort, we will just store the removed value in the now empty last node and no longer consider it as part of the heap.



Heaps: Demonstration: Removing the Root



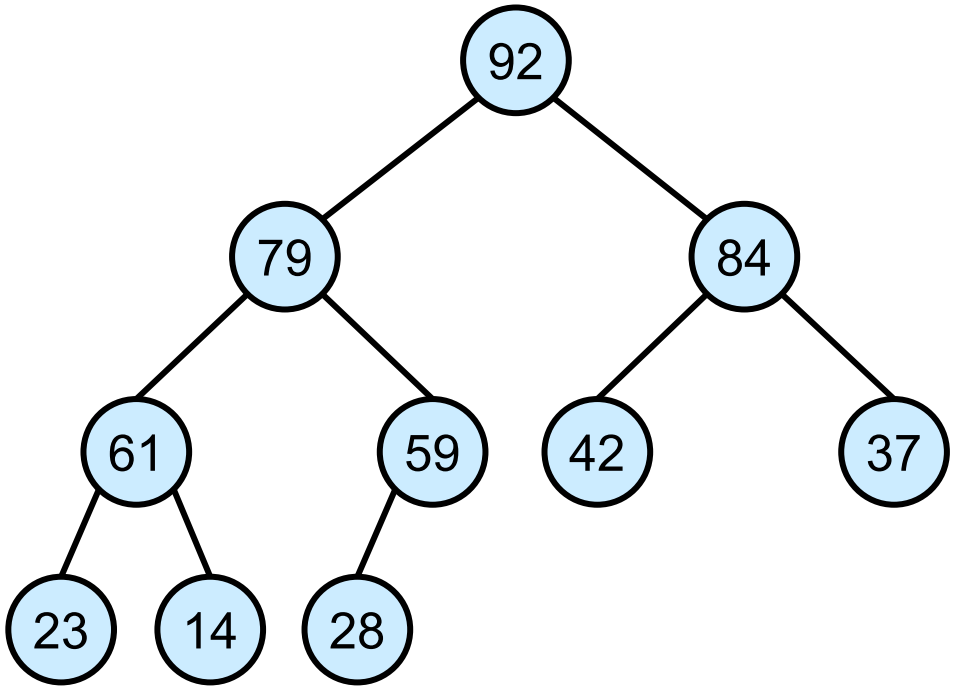
14	23	28	42	59	37	79	92	61	84
23	42	28	61	59	37	79	92	84	14
28	42	37	61	59	84	79	92	23	14
37	42	79	61	59	84	92	28	23	14
42	59	79	61	92	84	37	28	23	14
59	61	79	84	92	42	37	28	23	14
61	84	79	92	59	42	37	28	23	14
79	84	92	61	59	42	37	28	23	14
84	92	79	61	59	42	37	28	23	14
92	84	79	61	59	42	37	28	23	14
92	84	79	61	59	42	37	28	23	14

Heaps: Building a Heap

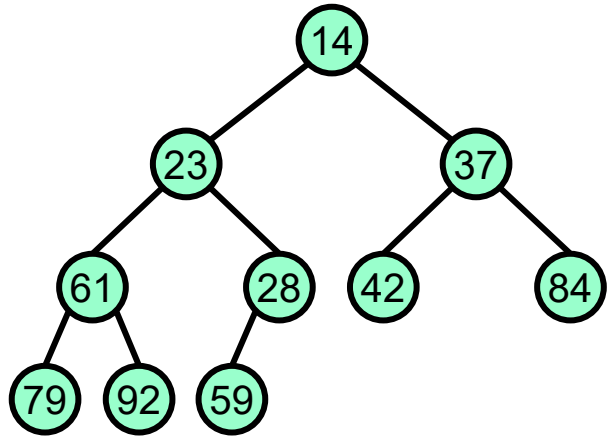
- How to get to a heap from an unordered array of values?

- Let's assume that for a certain node, we knew that both its subtrees are already heaps: In this case, we can apply the same procedure which we used before to trickle down the nodes.
- Thus, we just start at the last node which has at least one child node and then work our way up to the root while making sure for each node, that everything below (its subtrees) are nice heaps.

Heaps: Demonstration: Building a Heap



92	79	84	61	59	42	37	23	14	28
92	79	84	61	28	42	37	23	14	59
92	79	84	14	28	42	37	23	61	59
92	79	37	14	28	42	84	23	61	59
92	14	37	23	28	42	84	79	61	59
14	23	37	61	28	42	84	79	92	59



Heaps: Sorting Algorithm

1) Transform input data into heap.
2) Successively remove the root element and reorganize the remaining heap.

Exercise 5, Task 1.1: Building a Max-Heap

Given the input array A = [46, 77, 55, 38, 41, 85], what is the state of A after the algorithm BuildHeap(A, 6) that builds a max-heap has been executed?

- A. [85, 77, 55, 41, 38, 46]
- B. [38, 41, 46, 77, 55, 85]
- C. [85, 55, 77, 38, 41, 46]
- D. [85, 77, 55, 38, 41, 46]

→ **D**

Exercise 5, Task 1.2: Running Heap Sort

A heap has an array representation and a tree representation (nearly complete binary tree). Run the HeapSort algorithm on the array A = [11, 0, 9, 19, 8, 1, 5, 13, 18, 7] to sort elements of A in an ascending order. Show states of tree representation and array representation when the state of A changes.

Remark: The first sentence of the task description is actually not entirely correct. A heap **is** a nearly complete binary tree. This tree can be represented as an array (they are not on a par).

Exercise 5, Task 1.3: Implementing Heap Sort

Implement Heapify(A,i,s), BuildHeap(A,n) and HeapSort(A,n) in C. Use the array A in task 1.2 as the input array to print A's array representation.

Remark: In the sample solution, a swap function is used involving pointers (which is a feature of C that we did not yet discuss). Just ignore this for the moment, we will come back to pointers soon.

Exercise 5, Task 1.4: Analysis of Heapify

How many times the function Heapify has been executed in HeapSort to completely sort an array?

- A. n/2
- B. (3n-2)/2
- C. n-1
- D. n

→ **B**

Exercise 5, Task 1.5: Worst Case Complexity of Heap Sort

What is the worst case time complexity of HeapSort?

A. $O(n)$

B. $O(2n)$

C. $O(n \cdot \log(n))$

D. $O(n^2)$

→ **C**

Heapifying a Tree / Array Into a Min Heap in C Code

```
void heapify(int A[], int current_node, int length) {
    int left_child = get_left_child(current_node);
    int right_child = get_right_child(current_node);

    int smallest_node = current_node;
    if (left_child < length && A[left_child] < A[smallest_node]) {
        smallest_node = left_child;
    }
    if (right_child < length && A[right_child] < A[smallest_node]) {
        smallest_node = right_child;
    }

    if (smallest_node != current_node) {
        swap(A, current_node, smallest_node);
        heapify(A, smallest_node, length);
    }
}
```

get the array position of the left and right child of the current node

detect which of the three nodes (parent, left child, right child) is the smallest

if necessary, swap current node with smallest node and continue recursively upwards the tree structure

Comparison of Heapifying Code for Min Heap vs. Max Heap

```
void min_heapify(int A[], int i, int n) {
    int min = i;
    int l = lchild(i);
    int r = rchild(i);

    if (l < n && A[l] < A[min]) {
        min = l;
    }
    if (r < n && A[r] < A[min]) {
        min = r;
    }

    if (min != i) {
        swap(A, i, min);
        heapify(A, min, n);
    }
}
```

```
void max_heapify(int A[], int i, int n) {
    int max = i;
    int l = lchild(i);
    int r = rchild(i);

    if (l < n && A[l] > A[max]) {
        max = l;
    }
    if (r < n && A[r] > A[max]) {
        max = r;
    }

    if (max != i) {
        swap(A, i, max);
        heapify(A, max, n);
    }
}
```

Building a Heap in C Code

```
void buildHeap(int A[], int length) {
    int i;
    for (i = length/2; i >= 0; i--) {
        heapify(A, i, length);
    }
}
```

Remark: The same code is used irrespective of whether a min heap or a max heap shall be built; only the heapify function on line 4 is replaced with the respective version.

Heapsort in C Code

```
void heapSort(int A[], int length) {
    buildHeap(A, length);

    int last_node_still_in_heap = length;
    int i;
    for (i = length-1; i > 0; i--) {
        swap(A, i, 0);
        last_node_still_in_heap--;
        heapify(A, 0, last_node_still_in_heap);
    }
}
```

transform the input array into a valid heap

successively take out the root and place it to the position which got free by removing the node (i.e. node next to the last node still considered a part of the heap).

Building a Heap: Comprehension Question

Why do we iterate from ⌊n/2⌋ to 1 and not from 1 to ⌊n/2⌋ in the BuildHeap algorithm?

(Does this even matter?)

Why do we iterate only from ⌊n/2⌋ and not until n?

Algo: Heapify(A,i,s)

```
m = i;
l = Left(i);
r = Right(i);
if l≤s ∧ A[l]>A[m] then m = l;
if r≤s ∧ A[r]>A[m] then m = r;
if i≠m then
    exchange A[i] and A[m];
    Heapify(A,m,s);
```

Algo: BuildHeap(A)

```
for i = ⌊n/2⌋ to 1 do Heapify(A,i,n);
```

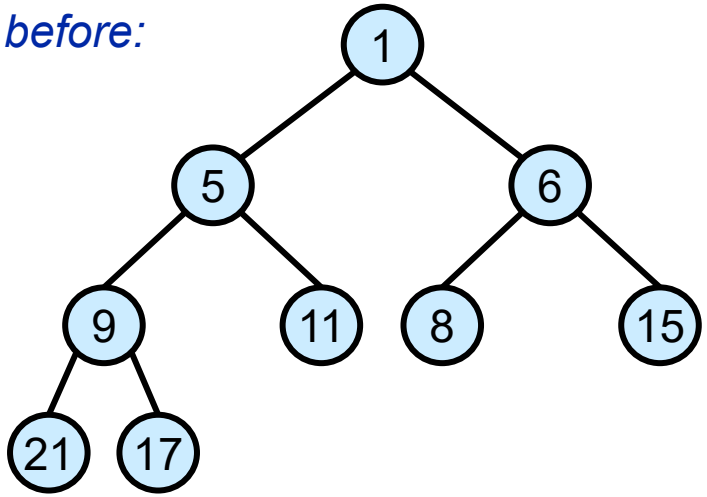
Heaps: Inserting a Node Into a Min Heap

A new node can be inserted into a min heap by performing the following steps:

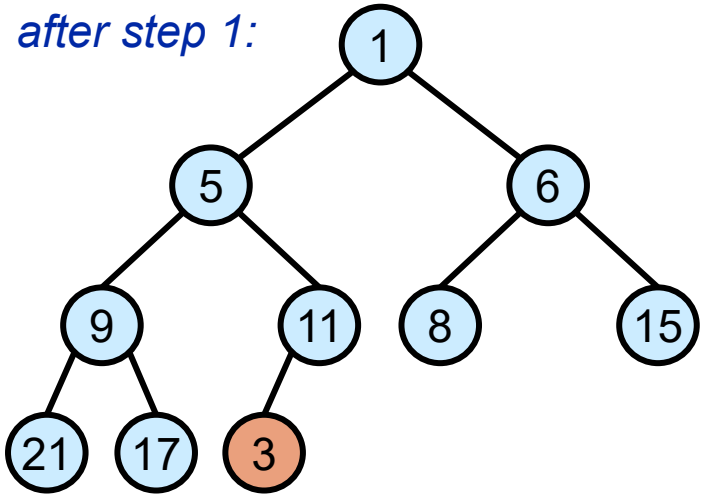
1) Insert the new node/element to the end of the heap/array (which will break the heap property in general)
2) To restore the heap property, sift *up* the new node by swapping it with its parent node as long as the parent node is bigger than the new node.

Example:

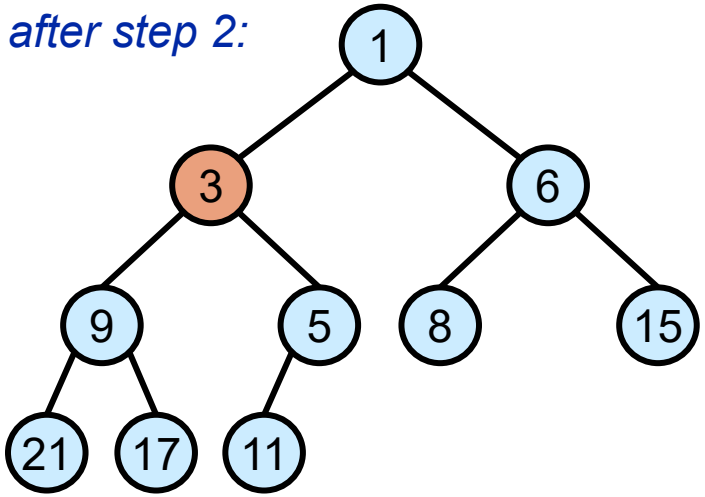
before:



after step 1:



after step 2:



Heaps: Deleting any Node From a Heap

Exercise: How can an arbitrary node, specified by its node value, be deleted from a heap?

Removing a particular node is just a generalized form of removing the root (i.e. we have to do the same thing which we do when removing the root but just applied to that particular node):

1) Find the node in the tree (if not given by index, this will require searching the whole tree in the worst case).
2) Delete the node from the heap / array (leaving a «hole» in the tree temporarily)
3) Place the node from the last position in the heap (temporarily) at the position of the node removed in the previous step. (This will make the tree a nearly complete binary tree again, but the heap conditions probably are not fulfilled yet.)
4) Repair the heap by sifting down the newly inserted node (i.e. apply the heapify algorithm)

Note: Deleting a particular node (other than the root node) from a Heap is typically not an expected operation and usually not supported by this data structure. If you feel it is necessary to delete a node other than the root from a heap, this is a very strong indication that you most probably should use another data structure which is better suited for your requirement.

Asymptotic Running Times of Heap Operations

- Heapify: O(log(n))
- Build heap: O(n · log(n)) *(note: this is an upper bound; it can be shown that the tight bound is Θ(n))*
- Remove root and re-heapify: O(log(n))
- Insert and re-heapify: O(log(n))
- Heapsort: O(n · log(n))

Heaps: Application: Priority Queue

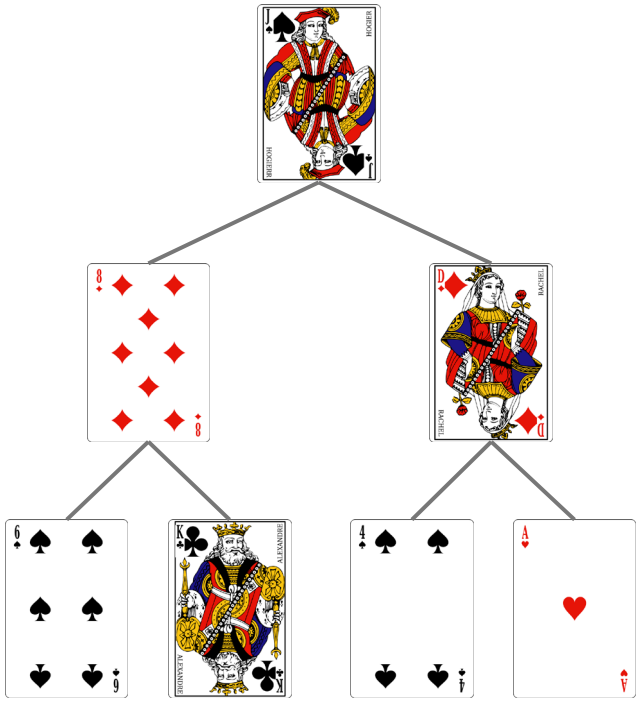
A very common application of a heap is the implementation of a priority queue (to the point that «priority queue» is sometimes treated almost synonymous to «heap»).

A priority queue is an abstract data type (ADT) which can be seen as an extension of the queue ADT where each element has a priority associated with it.

Elements with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.

A priority queue is used in the implementation of the famous Dijkstra algorithm which will be treated at a later point in this course.

Heaps, Heapsort: DIY / Game



1) Form groups of two to three people.
2) Get a deck of playing cards.
3) (If necessary: Familiarize yourself with the ordering of the cards.)
4) Shuffle the cards.
5) Lay out 12 cards in form of a binary three before you on the desk.
   (If you got the Joker, replace it through another card.)
6) Transform the card into a max- or min-heap.
7) Get a sorted set of cards by successively removing the root element.

8) If you feel comfortable with heapsort or if you get bored: Take 8 random cards of your deck and perform bubble sort, selection sort, insertion sort, quicksort, mergesort, … on it.

Heaps: Additional Practice

Additional exercises for practice can be found here:

https://h5p.org/node/457305

Additional Exercise: Time Complexities of Heap Operations

What is the minimal reachable asymptotic time complexity for the following operations performed on a binary heap with n nodes? Does it matter whether we're dealing with a min-heap or a max-heap?

a) Finding the k-th biggest node value in the heap

b) Printing the k smallest node values in the heap

c) Print all node values smaller than a given value x

a) The k-th biggest node value can be found in a max-heap by successively removing the root node (which is always the biggest node) and fixing (heapifying) the heap afterwards. This has to be done k times to find the k-th biggest value. Since the heapify operation runs in log(n) asymptotic time and has to be executed k times, the asymptotic complexity if the whole operation will be **O(k · log(n))**. Since k can take any value up to n, the asymptotic complexity could also be stated as **O(n · log(n))**. In a min-heap, the k-th biggest node value can be found by removing the root n − k times and thus the asymptotic complexity will be O((n − k) · log(n)) or O(n · log(n)) respectively. (Note: It is possible to find the k-th biggest value in an array in O(n) time using partitioning (QuickSelect algorithm), i.e. without making use of the heap properties.)

b) The k smallest node values can be printed in **O(k · log(n))** or **O(n · log(n))** time applying the procedure outlined in subtask a) independently on whether a min or max heap is given. (Again, it would be possible in linear time when an adaptation of QuickSelect is used.)

c) To achieve this task, all nodes of the heap have to be visited since the respective values could potentially be at (almost) all positions. On average and in the worst case, this can be done in **O(n)** time in both a min and max heap by iterating over the heap and printing all node values which fulfill the given condition. In the best case, if x is the minimum or maximum, the task can be achieved in **O(1)** time for the respective type of heap.

Quicksort

- Principles
- Partitioning

Quicksort: Principle

1) **Pivot selection:** Choose an element from the input array which should be used for dividing it into two subarrays; this element is called the pivot element (denoted with x here). In the example below, the last element of the array is choosen as pivot.

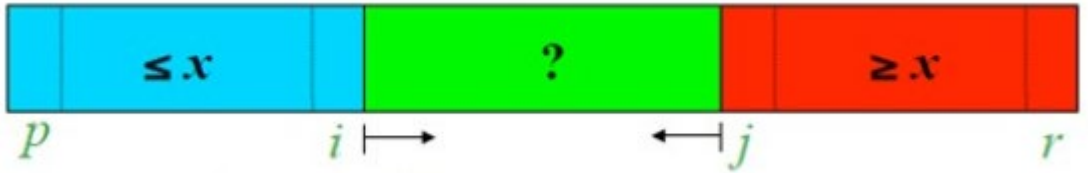
2) **Divide:** partition the array into two subarrays such that elements in the left part are smaller or equal to the elements in the right part:

3) **Conquer:** recursively apply the steps 1 and 2 (pivot selection and partitioning) on the two subarrays.

Partitioning Algorithms: Lomuto / Hoare

There are two main ways how the partitioning of the array in quicksort can be done:

- **Hoare's Algorithm**:
  - Grows two regions: A[p..i] from left to right and A[j..r] from right to left



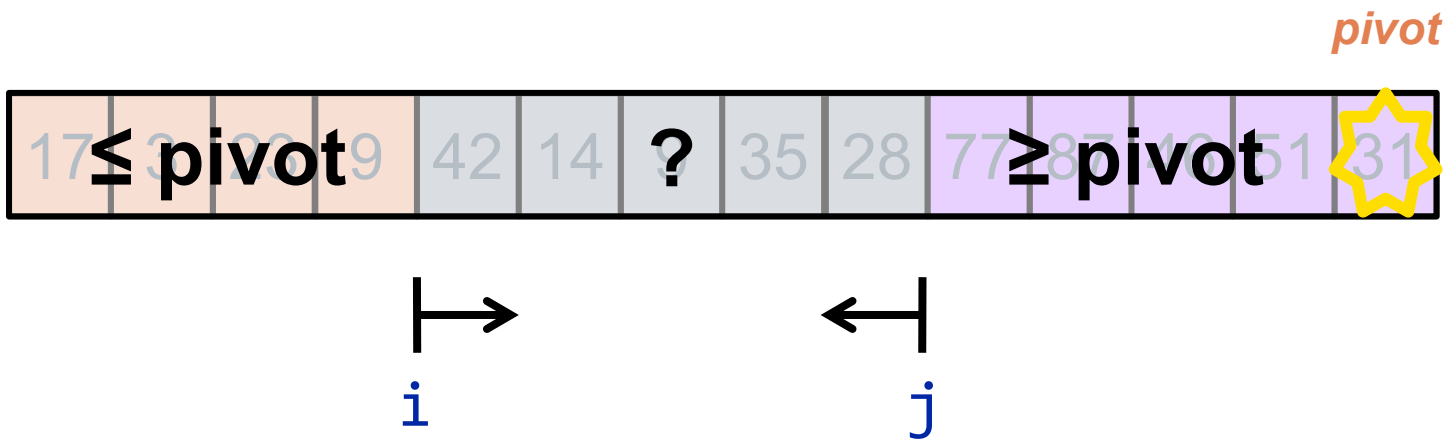
- **Lomuto's Algorithm**:
  - Works with a pivot; grows two regions from left: A[p..i] and A[i..j]; usually slower than Hoare's algorithm because it does three times more swaps on average

Hoare Partitioning Algorithm

Algo: HoarePartition(A,l,r)

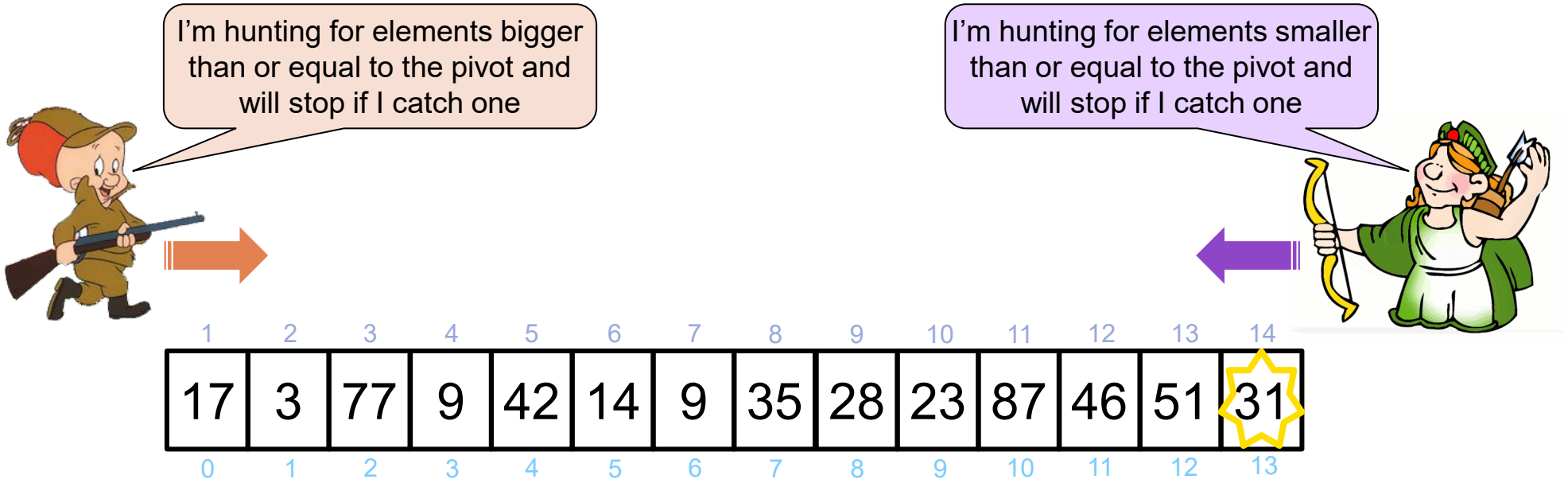
```
x = A[r]; i = l-1; j = r+1;
while true do
    repeat j = j-1 until A[j]≤x;
    repeat i = i+1 until A[i]≥x;
    if i<j then exchange A[i] and A[j] else return i;
```

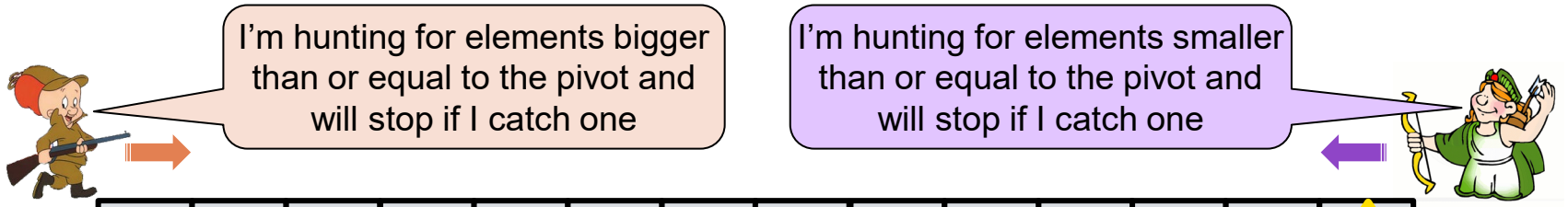
- The algorithm selects the rightmost element as the pivot x.
- It then grows two regions: one from left to right and one from right to left.



Hoare Partitioning: The Story of Two Hunters

Two hunters are marching along an array. One hunter starts at the left at a position before the first element and one hunter starts at the right at a position after the last element in the array. The left hunter will advance to the right (towards higher indices) and the right hunter will advance to the left (towards smaller indices). The hunters embark upon the following game: First, the right hunter advances to the right as long until he finds an element bigger than or equal to the pivot. When he found one, he will stop at once. Afterwards it's the turn of the left hunter who will do the same but looking for elements smaller than or equal to the pivot. When both hunters have stopped and they did not yet meet, they swap the elements they have found. The continue with this process until they meet which is when the game ends and they return the index of their meeting point.





Before the first round of the game:

17	3	77	9	42	14	9	28	35	23	87	46	51	31
1	2	3	4	5	6	7	8	9	10	11	12	13	14

After the first round of the game:

17	3	31	9	42	14	9	28	35	23	87	46	51	77

After the second round of the game:

17	3	31	9	23	14	9	28	35	42	87	46	51	77

After the third and last round of the game:

→ **return 9**

17	3	31	9	23	14	9	28	35	42	87	46	51	77
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Algo: HoarePartition(A,l,r)

x = A[r]; i = l-1; j = r+1;
while *true* **do**
  repeat j = j-1 **until** $A[j] \leq x$;
  repeat i = i+1 **until** $A[i] \geq x$;
  if $i < j$ **then** exchange A[i] and A[j] **else return** i;

Hoare Partitioning: The Story of Two Hunters

```
int partitionHoare(int A[], int leftArrayBoundary, int rightArrayBoundary) {
    int pivot = A[rightArrayBoundary];
    int i = leftArrayBoundary - 1;
    int j = rightArrayBoundary + 1;
    while (true) {
        do {
            i = i + 1;
        } while (A[i] < pivot);
        do {
            j = j - 1;
        } while (A[j] > pivot);

        if (i < j) {
            swap(A, i, j);
        }
        else {
            return i;
        }
    }
}
```

Selection of pivot (last element in array) and defining the starting position for the left hunter (i) and the right hunter (j).

The left hunter adavances to the right until he finds a value smaller than the pivot. (Note that this block is equivalent to: `while(A[++i] < pivot);`)

The right huntress adavances to the left until she finds a value bigger than the pivot. (Note that this block is equivalent to: `while(A[--j] > pivot);`)

If the two hunters have not yet met, the numbers they have caught will be swapped.

If the two hunters have met, their meeting position is returned and the algorithm ends.

Lomuto Partitioning Algorithm

The algorithm returns the position / index of the pivot in partitioned array.

It's complexity is Θ(n).

Algo: LomutoPartition(A,l,r)

```
x = A[r];
i = l-1;
for j = l to r-1 do
    if A[j]≤x then
        i = i+1;
        exchange A[i] and A[j];
exchange A[i+1] and A[r];
return i+1;
```

- Comprehension question: What happens when the input array is sorted ascendingly?
- Additional exercise: What is the loop invariant of Lomuto partitioning?

Lomuto Partitioning Algorithm: Comprehension Question

Consider the Lomuto algorithm as shown alongside and in the lecture. Why is the exchange statement on line 7 (marked in colour) necessary?

It is necessary only to handle the case when the pivot is the smallest element in the input array and could be omitted otherwise.

Algorithm: LomutoPartition(A, l, r)

```
1  x = A[r];
2  i = l − 1;
3  for j = l to r − 1 do
4      if A[j] ≤ x then
5          i = i + 1;
6          exchange A[i] and A[j];
7  exchange A[i + 1] and A[r];
8  return i + 1;
```

Partitioning: Comprehension Question

When applied to one and the same array, will the Lomuto partitioning algorithm and the Hoare partitioning algorithm always yield the same result?

Quicksort: Comprehension Question

Quicksort has an asymptotic running time of $O(n \cdot \log(n))$ in the average case and thus is faster than algorithms like bubblesort, insertionsort or selectionsort.

This is achieved by the quicksort algorithm because it has to do considerably less comparisons of elements than the aforementioned algorithms (in the average case).

Question: How / where does quicksort save comparisons?

Quicksort needs less comparisons because of the **partitioning** it applies. All elements on the left (lower) side of a partitioning will never have to be compared with an element on the right (upper) side of the partitioning. This is not ensured in algorithms like bubblesort.

This also reveals that there is a very close relationship between binary search trees and quicksort. Actually, bottom line, they both apply exactly the same principle.

Exercise 5 – Task 3: Partitioning in Quicksort: Solution

```
int partition(int A[], int l, int r) {
     int x = A[l], i = l;
     for (int j = l + 1; j <= r; j++) {
          if (A[j] <= x) {
               i += 1;
               int t = A[i];
               A[i] = A[j];
               A[j] = t;
          }
     }
     int t = A[i];
     A[i] = A[l];
     A[l] = t;
     return i;
}
```

Algo: LomutoPartition(A,l,r)

```
x = A[r];
i = l-1;
for j = l to r-1 do
    if A[j]≤x then
        i = i+1;
        exchange A[i] and A[j];
exchange A[i+1] and A[r];
return i+1;
```

Exercise 5, Task 2.1: Running Partitioning Algorithms by Hand

Run quicksort algorithms using Lomuto partition and Hoare partition on the input array A = [11, 0, 9, 19, 8, 1, 5, 13, 18, 7]. Show the state of A when A changes.

Exercise 5, Task 2.2: Implementing Quicksort

Implement LomutoPartition(A,l,r), HoarePartition(A,l,r) and Quick-Sort(A,l,r) taught in class in C code. Use the array A of task 2.1 as the input to check the correctness of your implementation.

Exercise 5, Task 2.3: Comparing Heapsort and Quicksort

In which case, heap sort is asymptotically faster than quick sort?

- Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is n lg n, and the constant factors hidden in the n lg n notation are quite small. It also has the advantage of sorting in place.
- A good implementation of quicksort usually will beat heapsort in practice.

Exercise 5, Task 3: Choosing a Pivot for Quicksort

Partitioning plays a crucial role in Quicksort algorithm. What are the issues of always choosing the element at the position n = 2 of an array of n elements as the pivot?

Additional Exercise

What is / are the difference / differences between the quicksort and the heapsort algorithm?

Sorting Algorithms Overview

- Comparison of Sorting Algorithms
- Stable and Instable Sorting

Comparison of Sorting Algorithms

Algorithm	*Asymptotic Time Complexity*			*Additional space required*	*Stable?*
	best case	*average case*	*worst case*		
Bubble sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	(in place)	yes
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	(in place)	no
Insertion sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	(in place)	yes
Merge sort	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	$\Theta(n)$	yes
Heapsort	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	(in place)	no
Quicksort	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	$\Theta(n^2)$	$\Theta(\log(n))$	no

Comparison of Sorting Algorithms: Comprehension Questions

Are there other sorting algorithms which are even faster?

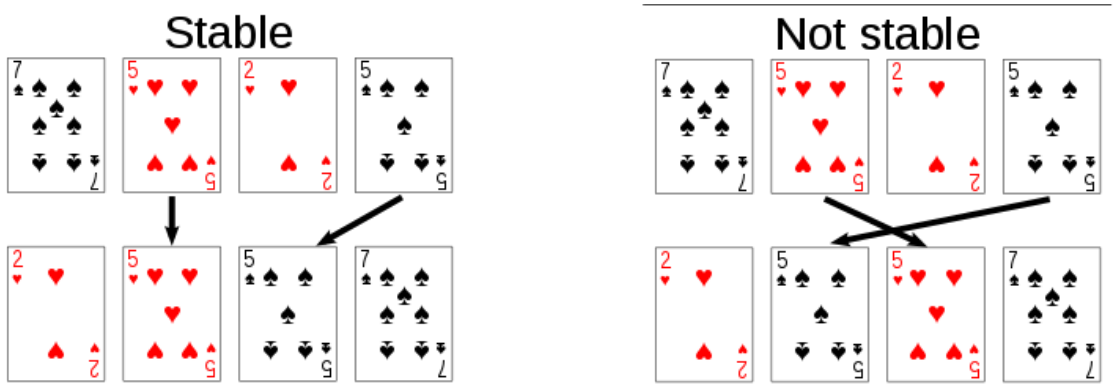
Which is the best sorting algorithm? Which one should I use?

Which algorithm will be applied when I call the `sort` method in Python?

Stable and Instable Sorting

A sorting algorithm is called stable if it does not change the order of elements with the same value. This is relevant for example when sorting data with several dimensions like for example an address list with names and dates of birth: If the list is sorted by name first and afterwards by date of birth, a stable sorting algorithm will preserve the sorted property of the names while an instable algorithm will not.

Example: Consider a hand of cards which shall be sorted by suits first and then by rank (number).



Note: Every sorting algorithm can be modified relatively easily to be stable.

Wrap-Up

- Summary
- Feedback
- Outlook
- Questions

Wrap-Up

– Summary

Outlook on Next Thursday's Lab Session

Next tutorial: Wednesday, 06.04.2022, 14.00 h, BIN 0.B.06

Topics:

- Review of Exercise 6
- Preview on Exercise 7
- …
- … (your wishes)

Questions?

Thank you for your attention.