



# Tutorial Session 5

Tuesday, 20<sup>th</sup> of March 2020

Discussion of Exercise 4, Preview on Exercise 5, Preparation for Midterm 1

Divide and Conquer, Recurrences

14.15 – 15.45

Y35-F-32



## Agenda

- Outlook on Midterm 1
- Discussion of Exercise 4 (incl. Divide and Conquer)
- Recurrences
- Loop Invariants
- Midterm Preparation
- (Preview on Exercise 5)



Universität  
Zürich<sup>UZH</sup>

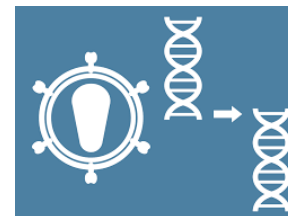
Institut für Informatik

Thank You!



Universität  
Zürich<sup>UZH</sup>

Institut für Medizinische Virologie



# OLAT

undefined

## Oops, something went wrong!

An error occurred and could not be solved automatically. If you want to contact the support team [support@olat.uzh.ch](mailto:support@olat.uzh.ch) please recall the following information

Please let us know what you did before you got this error message. Use the form below to describe what you did just before the error happened or what action triggered the error. It is of great importance to us to get this information to help finding and fixing the error as fast as possible, thank you!

You will receive an e-mail to the address you indicated in your OLAT profile.

[Back](#) [Send crash report](#) [Login](#)



INF\_20\_FS\_Informatics\_II

Tools

195052/CourseNode/101  
32/CourseNode/1013314



Wegen erhöhtem Zugriff steht Ihnen OLAT momentan nicht zur Verfügung.

Bitte versuchen Sie es zu einem späteren Zeitpunkt wieder.

Wir bitten um Ihr Verständnis.

Ihr OLAT-Team

## OLAT - Online Learning and Training

Wegen stark erhöhter Zugriffe auf OLAT kann es zu Ausfällen kommen. Wir bitten um Verständnis. Wenn möglich, greifen Sie zu Randzeiten (früh morgens oder abends) auf OLAT zu, damit sich die Last etwas verteilt. Wir arbeiten an einer nachhaltigen Lösung.

Wegen grösserer Systemauslastung kann es bis 48h dauern bis der Podcast (SWITCHcast) einer Veranstaltung im OLAT zur Verfügung steht.

Anmelden mit AAI-Konto

Anmelden mit OLAT Konto

Gastzugang

### OLAT Anmeldung

Bitte wählen Sie Ihre Hochschule.  
Für die Authentifizierung werden Sie weitergeleitet.

Wählen Sie die Organisation aus, der Sie angehören. ...

Login



# Outlook on Midterm 1

- General Information
- What to Expect
- Exam Strategy
- Remarks on the Cheat Sheet





## General Information: Topics and Allowed Auxiliary Materials

The midterm will take place at the originally scheduled time, but it will be conducted via OLAT.

- *Topics covered:* [SL01 to SL03](#), completely
- *Auxilliary materials:* the exam [open book](#), *no* foreign assistance is allowed, though

The style of the exam was announced to be comparable to earlier years.

Have a look at previous year's midterms which are available on the tutorial web page. Also, old assignments with solutions are available on the tutorial web page and can be used for additional practice.

The formula for calculating the grade from midterms and final stays in place.



## Exam Strategy

- Be ready (e.g. have cheat sheet ready, stable internet connection, suitable working space, ...)
- Start with tasks which you feel comfortable to solve.
- Make sure, you always at least **write something**. Note down observations and thoughts for example.
- If you struggle with formal notation, **use natural language** additionally (or alternatively).
- Never give up.

## Remarks on the Cheat Sheet / Books / Online-Resources / ...

The cheat sheet can be a [nice help](#). It saves you from learning stupid things by heart. It may protect from «blackout situations» where you suddenly can't remember things that you actually know.

*Although:*

- The cheat sheet is [useless](#) and can't save you, [if you don't understand](#) the content and [did not practice](#) its application.
- Preparing the cheat sheet may be a good preparation and repetition. I suggest, though, to [not spend too much time](#) into preparing the perfect «God cheat sheet» – [solving example tasks](#) (e.g. old midterms and assignments) is far more important and rewarding in my opinion.





## Review of Exercise 4

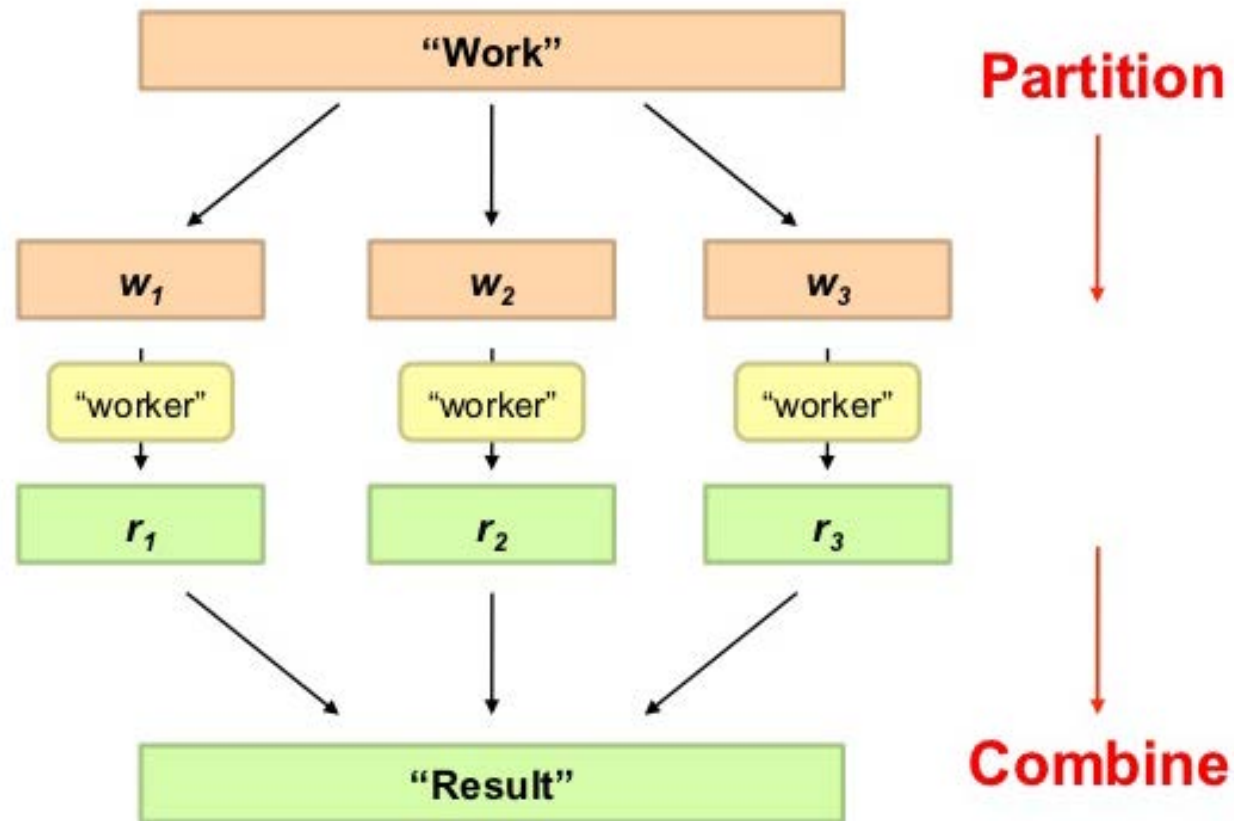
Divide and Conquer:

- Task 1: Finding the Element with Odd Occurrences Efficiently
- Task 2: Maximum-Subarray Algorithm

Recurrences:

- Task 3: Recurrence Tree and Substitution Method
- Task 4: Master Method

## Divide and Conquer Algorithms





## Exercise 4 – Task 1: Finding the Odd Element Efficiently

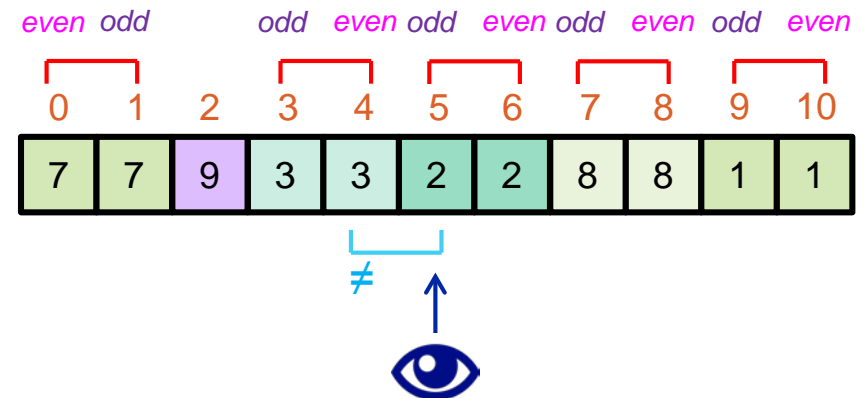
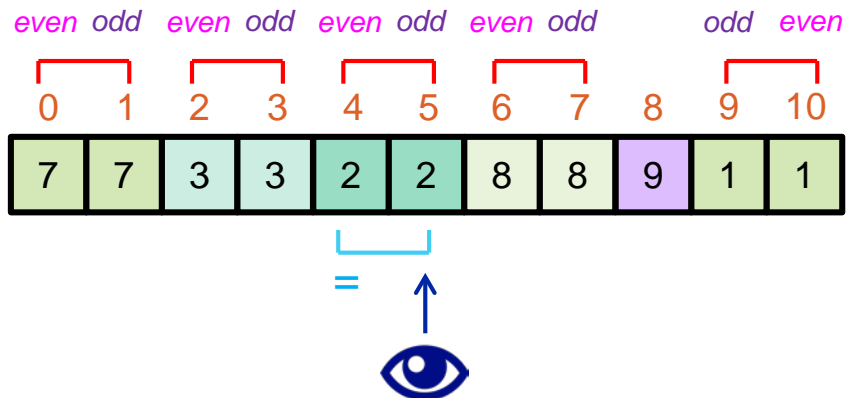
Given an array  $A[0..n - 1]$  of integers, every element has an even number of occurrences except one element with odd number of occurrences which we call odd element. An element with even occurrences appears in pairs in the array. Find an odd occurring element with an asymptotic complexity of  $O(\log n)$  using divide and conquer approach.

- a) Draw a tree to illustrate the process of finding the odd occurring element in the array  $A = [2, 2, 1, 1, 3, 3, 2, 2, 4, 4, 3, 1, 1]$ , according to your divide and conquer algorithm.
- b) Provide a C code for divide and conquer odd occurring element finder algorithm.

## Exercise 4 – Task 1: Finding the Odd Element Efficiently

The elements which are occurring an even time in the array appear **in pairs**. Their position index in the array always start at an **even index** and end at an **odd index** as long as the **single value with an odd occurrence** has not appeared in the array. The appearance of the **odd element** introduces an offset in this pattern such that **pairs** will start an **odd index** and end with an **even index** afterwards. When looking at some arbitrary element, we can thus evaluate whether the **odd element** has already appeared before or not:

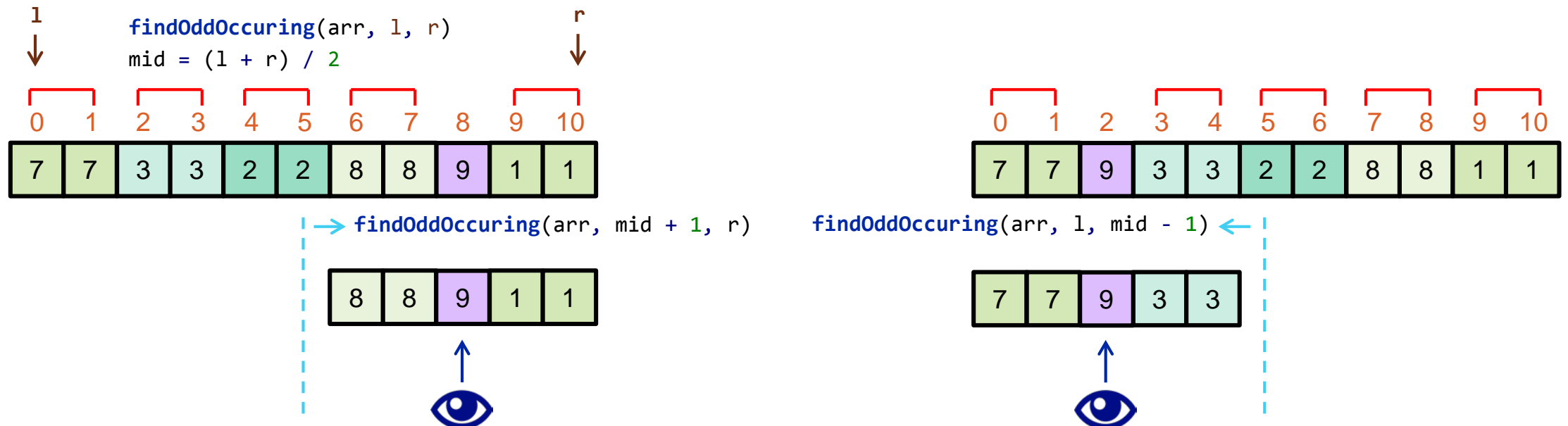
- if we're looking at an element with odd index the element to the left should be equal if the odd element has not yet occurred,
- if we're looking at an element with even index the element to the right should be equal if the odd element has not yet occurred.



## Exercise 4 – Task 1: Finding the Odd Element Efficiently

The technique to detect whether the odd element has already appeared in the array is now applied to perform a binary search pattern. We start in the middle of the array and detect whether the odd element has already appeared:

- If it hasn't, we continue recursively in the right half of the array.
- If it has, we continue recursively in the left half of the array.



## Exercise 4 – Task 1: Finding the Odd Element Efficiently

```

int findOddOccuring(int arr[], int l, int r){
base case { if (l == r) {
            return l;
          }

    int mid = (l + r) / 2; ← find middle element of (current sub-) array
    if (mid % 2 == 1) { // if mid is odd
        if (arr[mid] == arr[mid - 1]) ← look to left of middle element
            return findOddOccuring(arr, mid + 1, r); ← recursively continue with right
        else                                     half of array arr[]
            return findOddOccuring(arr, l, mid - 1); ← recursively continue with left
        }                                     half of array arr[]
    else { // mid is even
        if (arr[mid] == arr[mid + 1]) ← look to right of middle element
            return findOddOccuring(arr, mid + 2, r);
        else
            return findOddOccuring(arr, l, mid);
    }
}

```

Diagram annotations:

- A blue arrow points from the text "left boundary" to the variable `l` in the function signature.
- A purple arrow points from the text "right boundary" to the variable `r` in the function signature.
- A brown arrow points from the text "find middle element of (current sub-) array" to the calculation `(l + r) / 2`.
- A green arrow points from the text "look to left of middle element" to the condition `arr[mid] == arr[mid - 1]`.
- A blue arrow points from the text "recursively continue with right half of array arr[]" to the recursive call `findOddOccuring(arr, mid + 1, r)`.
- A blue arrow points from the text "recursively continue with left half of array arr[]" to the recursive call `findOddOccuring(arr, l, mid - 1)`.
- A green arrow points from the text "look to right of middle element" to the condition `arr[mid] == arr[mid + 1]`.
- A blue bracket groups the two recursive calls in the even case, with the text "either continue recursively with left or right half of array arr[]" next to it.



## Exercise 4 – Task 2: Maximum-Subarray Algorithm

The maximum-subarray algorithm finds the contiguous subarray that has the largest sum within an unsorted array  $A[0...n-1]$  of integers. For example, for array  $A = [-2, -5, 6, -2, -3, 1, 5]$ , the maximum subarray is  $[6, -2, -3, 1, 5]$ .

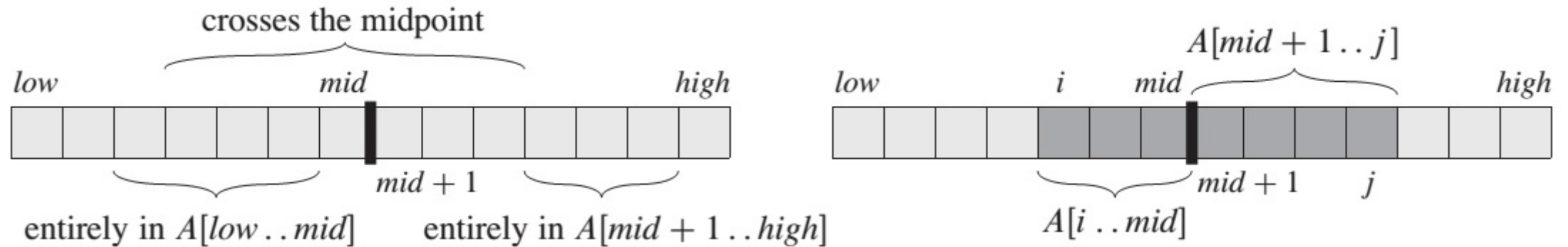
The algorithm works as follows:

Firstly, it divides the input array into two equal partitions: I ( $A[0]...A[mid]$ ) and II ( $A[mid+1]...A[n-1]$ ). Afterwards, it calls itself recursively on both partitions to find the maximum subarray of each partition. The combination step decides the maximum-subarray by comparing three arrays: the maximum-subarray from the left part, the maximum-subarray from the right part, and the maximum-subarray that overlaps the middle. The maximum-subarray that overlaps the middle is determined by considering all elements to the left and all elements to the right of the middle.

- Based on the above algorithm description, draw a tree that illustrates the process of determining the maximum subarray in array  $A = [-1, 2, -3, 4, 3, -5, 1, 5]$ .
- Provide a C code for maximum-subarray algorithm.
- Calculate its asymptotic tight bound.

## Exercise 4 – Task 2: Maximum-Subarray Algorithm

There are three possible configurations where the maximum subarray can be located in the input array:



## Exercise 4 – Task 2: Maximum-Subarray Algorithm

-1	2	-3	4	3	-5	1	5
----	---	----	---	---	----	---	---

-1	2	-3	4
----	---	----	---

3	-5	1	5
---	----	---	---

-1	2
----	---

-3	4
----	---

3	-5
---	----

1	5
---	---

-1
----

2
---

-3
----

4
---

3
---

-5
----

1
---

5
---

$max = -1$

$max = 2$

$max = -3$

$max = 4$

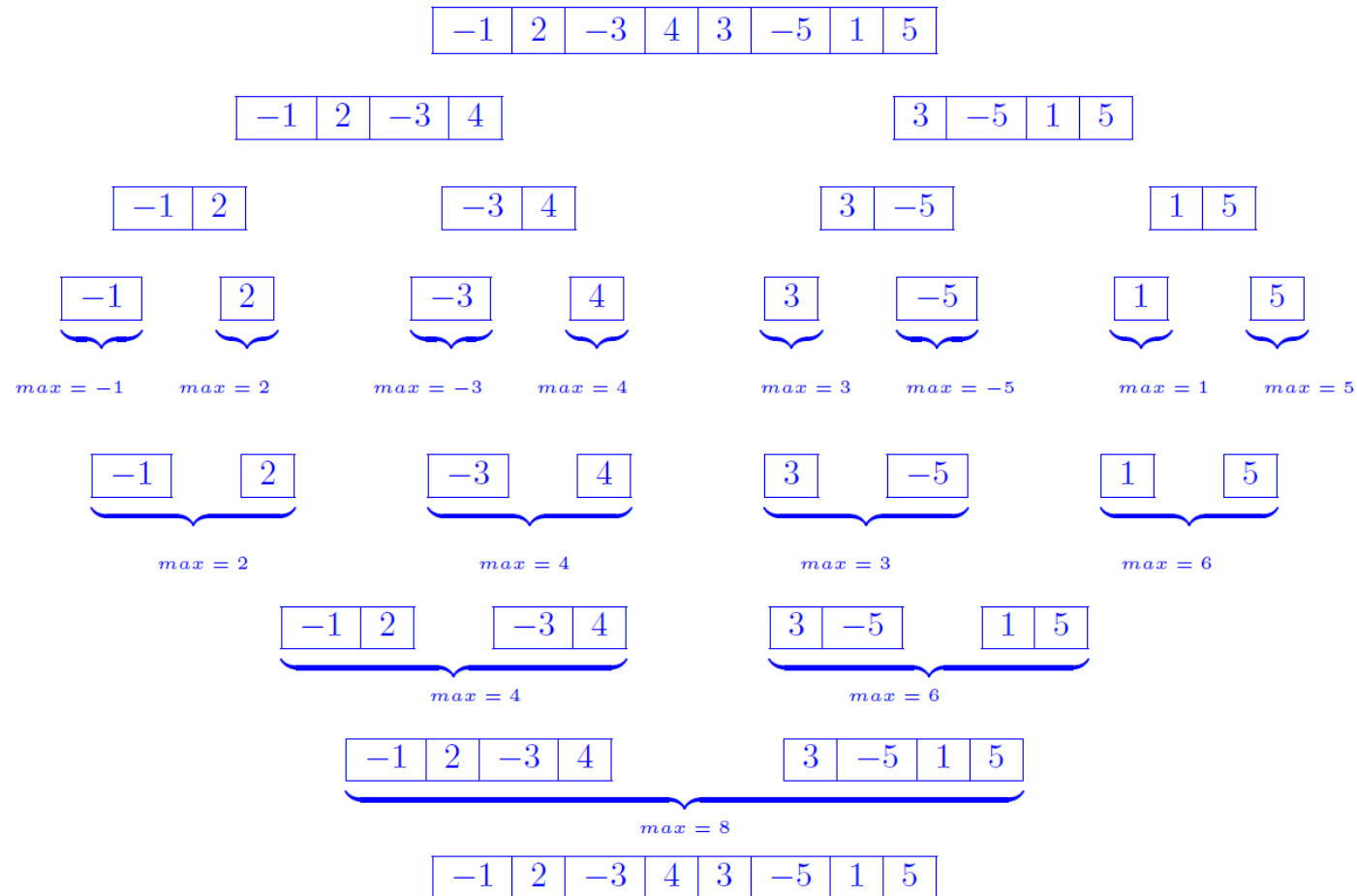
$max = 3$

$max = -5$

$max = 1$

$max = 5$

## Exercise 4 – Task 2: Maximum-Subarray Algorithm



## Exercise 4 – Task 2: Maximum-Subarray Algorithm

```
int maxOverlapArraySum(int arr[], int l, int m, int r) {
    int sum = 0, i = 0;
    int left_sum = -50000;
    for (i = m; i >= l; i--) {
        sum = sum + arr[i];
        if (sum > left_sum) {
            left_sum = sum;
        }
    }
    sum = 0, i = 0;
    int right_sum = -50000;
    for (i = m+1; i <= r; i++) {
        sum = sum + arr[i];
        if (sum > right_sum) {
            right_sum = sum;
        }
    }
    return left_sum + right_sum;
}
```

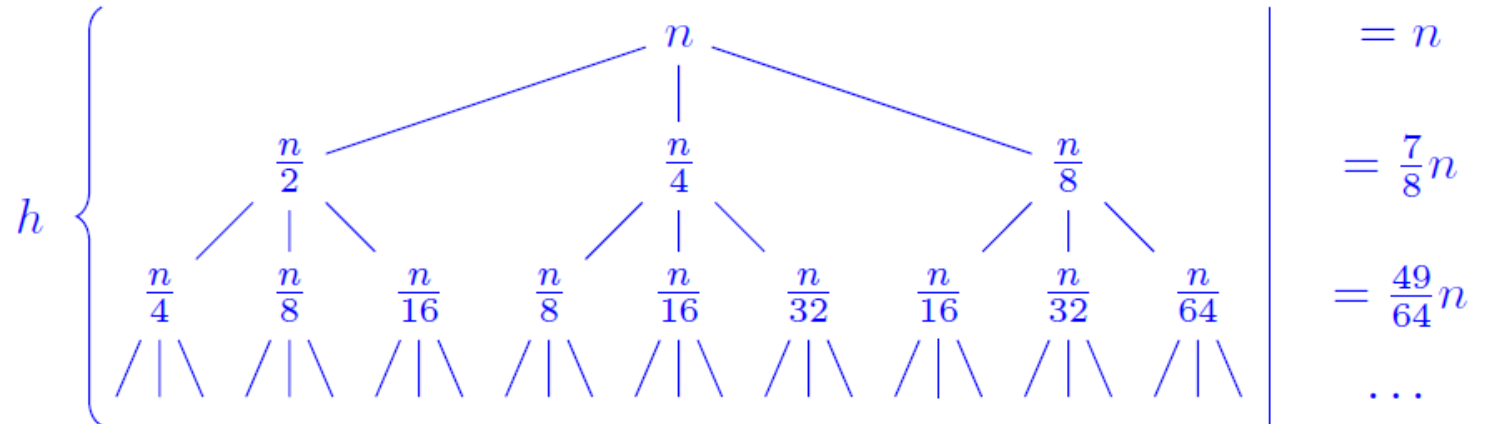
```
int maxSubArraySum(int arr[], int l, int r) {
    if (l == r)
        return arr[l];
    int m = (l + r) / 2;
    int leftArraySum = maxSubArraySum(arr, l, m);
    int rightArraySum = maxSubArraySum(arr, m+1, r);
    int overlapArraySum = maxOverlapArraySum(arr, l, m, r);
    if (leftArraySum > rightArraySum &&
        leftArraySum > overlapArraySum) {
        return leftArraySum;
    } else if (rightArraySum > leftArraySum &&
        rightArraySum > overlapArraySum) {
        return rightArraySum;
    }
    return overlapArraySum;
}
```

## Exercise 4 – Task 3a: Solving a Recurrence Using Recursion Tree

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + T(n/4) + T(n/8) + n & \text{if } n > 1 \end{cases}$$

The branches of the recursion tree are longer on the very left side (where  $n$  gets always divided by 2). To estimate the total work in its nodes, we will assume the tree has the height of the leftmost branch everywhere and this will yield an upper bound for the total work.

On the left side, the tree grows until  $h = \log_2(n)$ . Therefore an upper bound for the total work is:



$$T(n) < n \cdot \sum_{i=0}^{\log_2(n)} \left(\frac{7}{8}\right)^i$$



## Addendum for Tasks 3a, 4.2 and 4.5: Geometric Series

A summation of the form

$$\sum_{k=0}^n a^k = 1 + a^1 + a^2 + a^3 + \dots + a^n$$

where  $a$  is some real number with  $0 < a \neq 1$  and  $n$  is an integer with  $n > 0$  is called a (finite) geometric series.

This finite geometric series sums up to

$$\frac{a^{n+1} - 1}{a - 1} = \frac{1 - a^{n+1}}{1 - a}$$

(If  $a = 0$ , then the series sums just up to  $n + 1$ .)

The infinite geometric series  $\sum_{k=1}^{\infty} a^k = 1 + a^1 + a^2 + a^3 + \dots$

converges to  $\frac{1}{1 - a}$  iff  $|a| < 1$ , otherwise it diverges.

## Exercise 4 – Task 3a: More Detailed Calculation for Guess

The sum  $\sum_{i=0}^h \left(\frac{7}{8}\right)^i$  is a finite geometric series, i.e. a sum of form  $\sum_{k=0}^x a^k$  where  $a = \frac{7}{8}$  and  $x = h = \log_2(n)$ .

We know that the geometric series sums up to  $\frac{1 - a^{x+1}}{1 - a}$ .

$$\text{Thus: } T(n) < n \cdot \sum_{i=0}^h \left(\frac{7}{8}\right)^i = n \cdot \frac{1 - \left(\frac{7}{8}\right)^{h+1}}{1 - \frac{7}{8}} = 8 \cdot n \cdot \left(1 - \left(\frac{7}{8}\right)^{h+1}\right) =$$

$$\begin{aligned} \text{With } h = \log_2(n) \text{ we get: } T(n) &< 8 \cdot n \cdot \left(1 - \left(\frac{7}{8}\right)^{h+1}\right) = 8 \cdot n \cdot \left(1 - \left(\frac{7}{8}\right)^{\log_2(n)} \cdot \left(\frac{7}{8}\right)^1\right) = \\ &= 8 \cdot n \cdot \left(1 - n^{\log_2\left(\frac{7}{8}\right)} \cdot \frac{7}{8}\right) = 8 \cdot n - 7 \cdot n \cdot n^{\log_2\left(\frac{7}{8}\right)} = 8 \cdot n - 7 \cdot n^{1 + \log_2\left(\frac{7}{8}\right)} \end{aligned}$$

Since  $\log_2(7/8) \approx -0.19$ ,  $n^{1+\log_2(7/8)} \approx n^{0.81}$  which is asymptotically smaller than the linear term of the first summand. Therefore,  $T(n)$  is in  $O(n)$ .

## Exercise 4 – Task 3b: Proofing a Recurrence Using Substitution

### Inductive Step

- $T(n) \leq cn \implies T(n/2) + T(n/4) + T(n/8) + n \leq c\frac{n}{2} + c\frac{n}{4} + c\frac{n}{8} + n$
- ?  $c\frac{n}{2} + c\frac{n}{4} + c\frac{n}{8} + n \leq cn$

### Proof

- $c\frac{n}{2} + c\frac{n}{4} + c\frac{n}{8} + n \leq cn$
- dividing both parts by  $n$  we get  $c\frac{1}{2} + c\frac{1}{4} + c\frac{1}{8} + 1 \leq c$
- $\frac{1}{8}c \geq 1 \implies c \geq 8$

### Asymptotic Complexity

$$T(n) = O(n), \text{ for } c \geq 8$$

## Exercise 4 – Task 4: Master Method

1.  $T(n) = 3 \cdot T(n/9) + 32 \cdot \sqrt[2]{n}$

$a = 3, b = 9, f(n) = 32 \cdot \text{sqrt}(n)$

$x = \log_b(a) = \log_9(3) = 0.5; n^x = n^{\log_9(3)} = n^{0.5} = \sqrt[2]{n}$

compare  $n^x$  to  $f(n)$ : same growth rate  $\rightarrow$  case 2

$T(n) \in \Theta(n^{0.5} \cdot \log(n))$

2.  $T(n) = T(\sqrt[3]{n}) + \log(n)$

master method not applicable: parsing fails

applying repeated substitution yields:  $T(n) \in O(\log(n))$

$$\begin{aligned} T(n) &= \log n + \log \sqrt[3]{n} + \log \sqrt[3]{\sqrt[3]{n}} + \dots \\ &= \log n + \frac{1}{3} \log n + \frac{1}{9} \log n + \dots \\ &= \left( \sum_{k=0}^{\infty} \left(\frac{1}{3}\right)^k \right) \log n \\ &= \frac{3}{2} \log n \end{aligned}$$



## Repeated Substitution: Tasks 4.2 and 4.5 of Exercise 4

*Expanded recurrence*

*Term currently to be expanded*

## Exercise 4 – Task 4: Master Method

3.  $T(n) = 15 \cdot T(n/4) + n^2$

$a = 15, b = 4, f(n) = n^2$

$x = \log_b(a) = \log_4(15) \approx 1.95; n^x = n^{\log_4(15)} \approx n^{1.95}$

compare  $n^x$  to  $f(n)$ :  $f(n)$  grows faster  $\rightarrow$  case 3

*Regularity check:* we need to find a constant  $c < 1$  such that  $a \cdot f(n/b) \leq c \cdot f(n)$  for any  $n > n_0$ .

$a \cdot f(n/b) = 15 \cdot f(n/4) = 15 \cdot n^2/4^2 = 15/16 \cdot n^2$

Thus we're looking for a  $c$  which fulfills:  $15/16 \cdot n^2 \leq c \cdot n^2$

The above equation is clearly fulfilled for  $c \geq 15/16 = 0.9375$  and any arbitrary  $n = n_0$ .

$T(n) \in \Theta(n^2)$



## Exercise 4 – Task 4: Master Method

4.  $T(n) = 2 \cdot T(n - 2) + n$

master method not applicable: parsing fails

applying repeated substitution yields:  $T(n) \in O(n \cdot (\text{sqrt}(2))^n)$

$$\begin{aligned} T(n) &= 2T(n - 2) + n \\ &= 2(2T(n - 4) + (n - 2)) + n \\ &= 4T(n - 4) + 3n - 4 \\ &= 4(2T(n - 6) + (n - 4)) + 3n - 4 = 8T(n - 6) + 7n - 20 \\ &= 8(2T(n - 8) + (n - 6)) + 7n - 20 = 16T(n - 8) + 15n - 68 \\ &= \dots \end{aligned}$$

$$\Rightarrow T(n) = 2^k T(n - 2k) + (2^k - 1)n - \sum_{i=0}^{k-1} 2^i \cdot 2i$$

$k$  grows until it reaches  $k = \lfloor \frac{n}{2} \rfloor$ , thus we have:

$$T(n) = (2^{\lfloor \frac{n}{2} \rfloor} - 1)n - \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor - 1} 2^i \cdot 2i \leq (2^{\lfloor \frac{n}{2} \rfloor} - 1)n - 2^{\lfloor \frac{n}{2} \rfloor - 1} 2 \cdot (\lfloor \frac{n}{2} \rfloor - 1) \in \Theta(n\sqrt{2}^n)$$

## Exercise 4 – Task 4: Master Method

5.  $T(n) = \log(n) + T(\sqrt[2]{n})$

master method not applicable: parsing fails

applying repeated substitution yields:  $T(n) \in O(\log(n))$

$$\begin{aligned} T(n) &= \log n + \log \sqrt{n} + \log \sqrt{\sqrt{n}} + \dots \\ &= \log n + \frac{1}{2} \log n + \frac{1}{4} \log n + \dots \\ &= \left( \sum_{k=0}^{\infty} \left(\frac{1}{2}\right)^k \right) \log n \\ &= 2 \log n \end{aligned}$$

## More Rigorous Justification for Task 4.4: Arithmetico-Geometric Series

By combining the arithmetic series and the geometric series by a term-by-term multiplication, we get a summation of the form

$$\sum_{k=0}^n (a_0 + k \cdot d) \cdot b^k = a_0 + (a_0 + d) \cdot b + (a_0 + 2 \cdot d) \cdot b^2 + (a_0 + 3 \cdot d) \cdot b^3 + \dots + (a_0 + n \cdot d) \cdot b^n$$

(with  $0 < b \neq 1$ ,  $d = \text{const}$  and an integer  $n > 0$ ).

This is called a (finite) **arithmetico-geometric series**. As it is written above (i.e. starting at  $k = 0$ ) sums up to

$$\frac{a_0}{1-b} - \frac{(a_0 + (n-1) \cdot d) \cdot b^n}{1-b} + \frac{d \cdot b \cdot (1-b^{n-1})}{(1-b)^2}$$

The infinite arithmetico-geometric series sums up to:

$$\sum_{k=0}^{\infty} (a_0 + k \cdot d) \cdot b^k = \frac{a_0}{1-b} + \frac{d \cdot b}{(1-b)^2}$$



# Recurrences

- Motivation
- Methods for solving recurrences
- Repeated substitution example solved
- Recursion tree example solved
- Master method recapitulation
- Master method recipe and example solved
- Substitution method / inductive proof example solved

## Motivation: Another Running Time Analysis Example

What is the asymptotic upper bound for the following C code fragment?

```
int functionH(int a, int n)
{
    if (n == 1) {
        return a;
    }
    else {
        return a + functionH(a, n/2);
    }
}
```

→  **$O(\log(n))$**

Follow-up question:

What is the result of  
**functionH(1, 100)?**

→ **7**



## Introduction Recurrences

The previous running time analysis example cannot be solved in the same way as this was done in earlier cases where we just determined the number of executions for each of the lines and multiplied this with the time needed (cost) for the respective line and add all together to get the total time needed.

Instead, here the time for execution of the function depends on the function itself: «The time needed to execute the function with parameter  $n$  is the time needed to execute the function for half the parameter size,  $n/2$ , plus some time for the non-recursive parts of the function (calculated as in the cases before)».

This kind of problem can be formulated as a **recurrence relation**:

$$T(n) = T(n/2) + 1.$$

We now need a method to rewrite this kind of equation into a non-recursive form where the  $T()$  function is only on one side of the equation. Recurrence relations can be thought of as mathematical **models of the behaviour of recursive functions** (in particular of divide and conquer algorithms).





## Introduction Recurrences

- Recurrence (relation): description of a function which is recursive:  $f(n) = g(f(n))$ .
- The phrase «**solving a recurrence**» means to find a way to calculate a result for any input value of the function without the need of recursive expansion (also called «to find a closed solution / expression»). Thus the recursively defined function should only be on one side of the equation.
- Solving recurrences is in a way similar to solving integrals: In both cases there is **no single one-works-for-all procedure**.
- There are recurrences which cannot be solved.  
(Example: most variants of the logistic map:  $T(n + 1) = r \cdot T(n) \cdot (1 - T(n))$ ).



## Recurrences: Interpretation Example

Consider the following recurrence:

$$T(n) = 2 \cdot T(n/5) + 3 \cdot T(n - 1) + 78 \cdot n$$

Explain what this means in light of algorithms.

## Running Time Analysis Example Revisited

Express the example C code function from before as a recurrence.

```
int functionH(int a, int n)
{
    if (n == 1) {
        return a;
    }
    else {
        return a + functionH(a, n/2);
    }
}
```

→  $T(n) = T(n/2) + 1$

## 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 they 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.)

## Repeated Substitution

*General application procedure:*

- 1) Perform a sequence of substitute – expand – substitute – expand – ... loops until you see a pattern.
  - 2) Find a general (still recursive) formula for the situation after the k-th substitution.
  - 3) Find out how many iterations have to be done in order to get to the base case.
  - 4) Fill in the number of iterations for the base case into the general formula for k-th substitution.
- Tends to be confusing (at least for me).
  - Can be done «in-line» or with a «helper row» (to probably avoid confusion).
  - Quickly becomes unsuitable for more complex cases, e.g. if multiple recursive calls are involved in the recurrence relation (an example recurrence where repeated substitution method would be a dead end street is for example:  $T(n) = T(n/3) + 2 \cdot T(n/3^2) + n$ ). In those cases, another method needs to be used, for example the recursion tree method (which is basically just a clever graphical depiction of the same thing helping to sum up terms).

## Repeated Substitution: Example

$$T(1) = 4; \quad T(n) = 2 \cdot T(n/2) + 4 \cdot n$$

iteration	Expanded recurrence <i>Fill in the <b>expanded term</b> from right side as a substitution for the <b>term that was expanded</b></i>	Term currently to be expanded <i>fill in the <b>argument</b> from left side everywhere you see <b>n</b> in the recurrence template; can be helpful to not simplify terms too early</i>
h =		
1	$T(n) = 2 \cdot \boxed{T(n/2)} + 4 \cdot n =$	$\boxed{T(n/2)} = 2 \cdot T((n/2)/2) + 4 \cdot (n/2)$
2	$= 2 \cdot [2 \cdot T((n/2)/2) + 4 \cdot (n/2)] + 4 \cdot n =$ $= 2^2 \cdot \boxed{T(n/2^2)} + 4 \cdot n + 4 \cdot n =$	$\boxed{T(n/2^2)} = 2 \cdot T((n/2^2)/2) + 4 \cdot (n/2^2)$
3	$= 2^2 \cdot [2 \cdot T((n/2^2)/2) + 4 \cdot (n/2^2)] + 8 \cdot n =$ $= 2^3 \cdot T(n/2^3) + 4 \cdot n + 4 \cdot n + 4 \cdot n = \dots$	$T(n) = 2^k \cdot T(n/2^k) + k \cdot (4 \cdot n)$

## Repeated Substitution: Example

- We found the general expression for the k-th iteration of substitution:  $T(n) = 2^k \cdot T(n/2^k) + k \cdot (4 \cdot n)$ .
- But when (at which  $k = x$ ) will we reach the base case  $T(1)$ ? (How often do we have to substitute / execute recursive calls?)
- This will be the case when  $T(n/2^x) = T(1)$ , i.e. when

$$n/2^x = 1$$

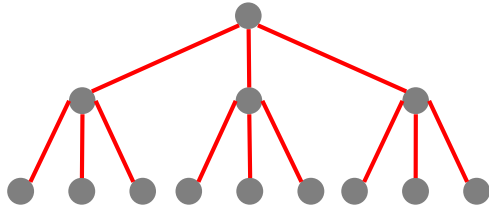
- From this:  $n = 2^x$ ,  $x = \log_2(n)$
- Now insert the  $x$  where we reach the base case into the **general expression** from above and then substitute the value given for  $T(1)$ :

$$\begin{aligned} T(n) &= 2^{\log_2(n)} \cdot T(1) + \log_2(n) \cdot (4 \cdot n) = \\ &= n \cdot 4 + \log_2(n) \cdot (4 \cdot n) = \\ &4 \cdot n + 4 \cdot n \cdot \log_2(n) \in \Theta(n \cdot \log(n)) \end{aligned}$$

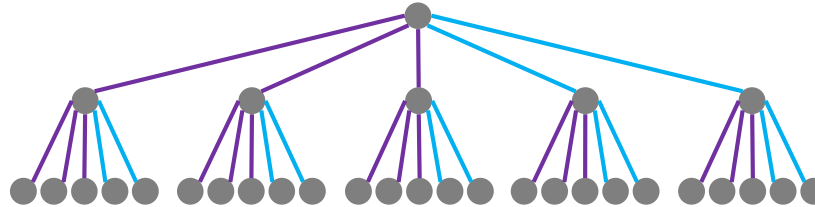
## Recursion Tree Method

- Basically the same idea as in the [repeated substitution](#) method, but [graphically visualized as a tree](#).
- Number of [recursive calls](#) corresponds to number of [branches per node](#).

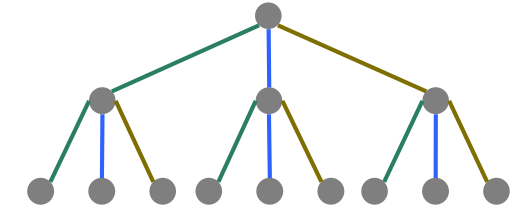
$$T(n) = 3 \cdot T(n/2)$$



$$T(n) = 3 \cdot T(n/4) + 2 \cdot T(n-6)$$



$$T(n) = 1 \cdot T(n-1) + 1 \cdot T(n-2) + 1 \cdot T(n-3)$$



- Values at [nodes](#) is [work](#) done [outside recursive calls](#) (at current recursive call)
- Sum of node values in the whole tree (down to the base case) is total work done.



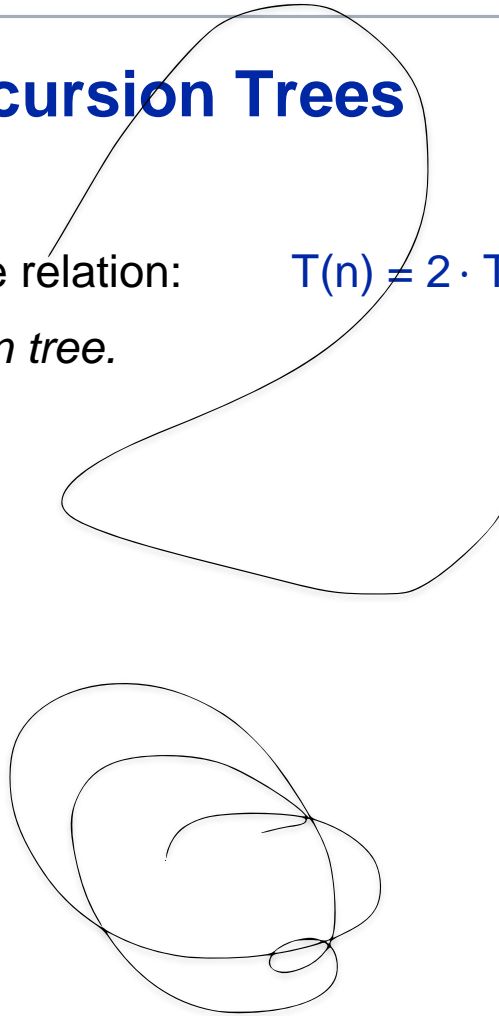


## Exercise: Drawing Recursion Trees

Consider the following recurrence relation:

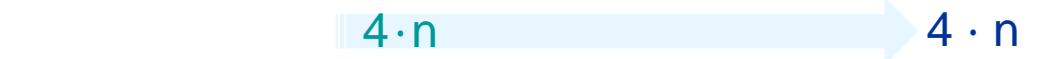



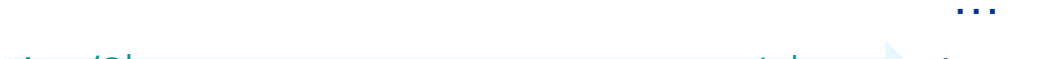

$$T(n) = 2 \cdot T(n / 4) + n^2 \cdot \sqrt{n} \cdot \log(n)$$

*Draw the corresponding recursion tree.*



## Recursion Tree Method: Example

Base case:  $T(1) = 4$ ;      branches per node:  $T(n) = 2 \cdot$       Recursive part:  $T(n/2)$       +      Work outside recursive call = values at nodes:  $4 \cdot n$

level	number of nodes	recursive call	non-recursive value	recursion tree	sum per row
0	1	$T(n)$	$4 \cdot n$		$4 \cdot n$
1	2	$T(n/2)$	$4 \cdot n/2$		$4 \cdot n$
2	$2^2$	$T(n/2^2)$	$4 \cdot n/2^2$		$4 \cdot n$
3	$2^3$	$T(n/2^3)$	$4 \cdot n/2^3$		$4 \cdot n$
...	...	...	...	...	...
k	$2^k$	$T(n/2^k)$	$4 \cdot n/2^k$		$4 \cdot n$
...	...	...	...	...	...
x	$2^x$	$T(1)$	4		$4 \cdot n$
					$\Sigma \dots$

## Recursion Tree Method: Example

- In order to get the **total work** done in the whole recursion tree, we have to **sum up the row sums**.
- Thus, we **need to know** the number of rows which is the same as the height / number of **levels of the tree**.
- Again, we have to find the number of iterations – or in this case: tree levels – needed to **get to the base case**. The respective considerations are analogous as for the repeated substitution method: The base case is reached when  $T(n/2^x) = T(1)$  i.e. when  $n/2^x = 1$  and from this we find:  $n = 2^x$ ,  $x = \log_2(n)$ .
- Now, we **sum up** the **row sums** and find the result:

$$\sum_{k=0}^x 4 \cdot n = \sum_{k=0}^{\log_2(n)} 4 \cdot n = 4 \cdot n \cdot \sum_{k=0}^{\log_2(n)} 1 = 4 \cdot n \cdot (\log_2(n) + 1)$$

$$4 \cdot n + 4 \cdot n \cdot \log_2(n) \in \Theta(n \cdot \log(n))$$



## Exercise: Recursion Tree Method

Consider the following recurrence relation:  $T(n) = 37 \cdot T(n / 5) + 21 \cdot T(n / 7) + 101 \cdot T(n / 9) + 3 \cdot n^2$ ;  $T(1) = 1$

*Explain what the recursion tree will look like with regard to its height.*

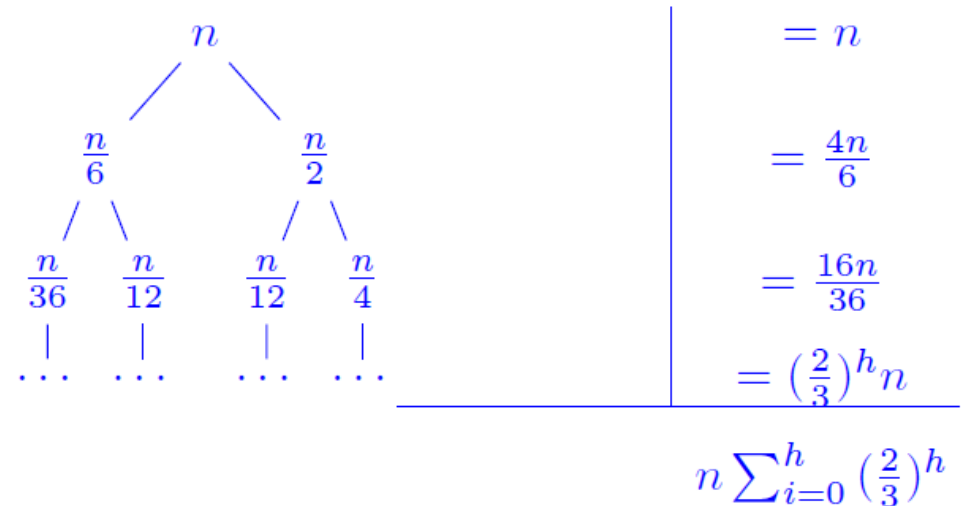
## Exercise: Another Recursion Tree Example

(Task 5a of assignment 2 from FS 2018) Solve the following recurrence using the tree method:

$$T(n) = T(n/6) + T(n/2) + n$$

$$T(1) = 1$$

$$h = \log_2 n$$



Longest branch on the right. Tree grows until  $\left(\frac{1}{2}\right)^h n = 1 \implies h = \log_2 n$   
 Guess:  $O(n)$



## 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

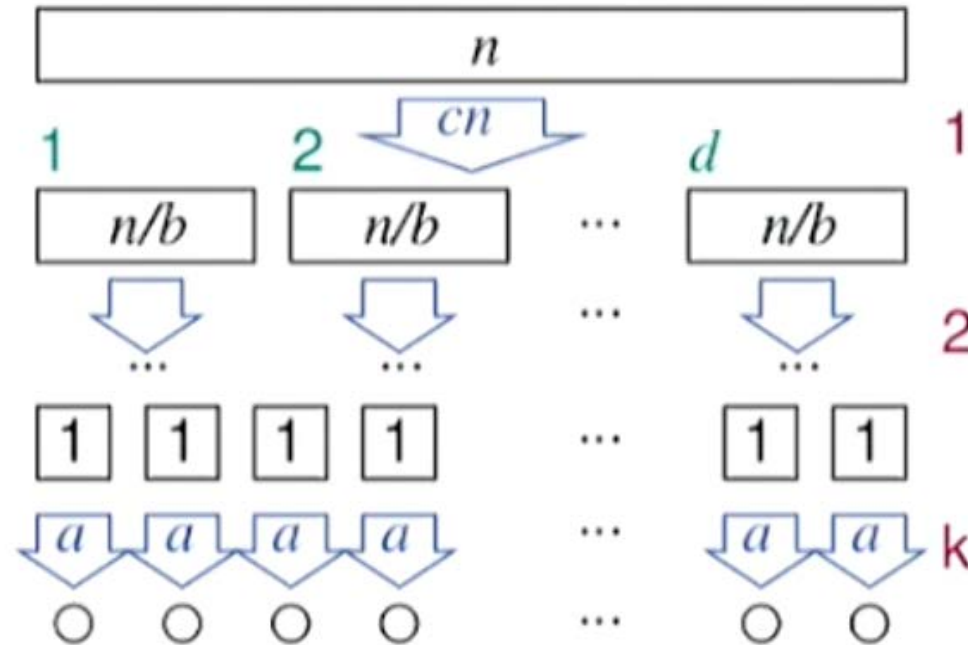
$$T(n) = a \cdot T(n/b) + f(n)$$

- $n$  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

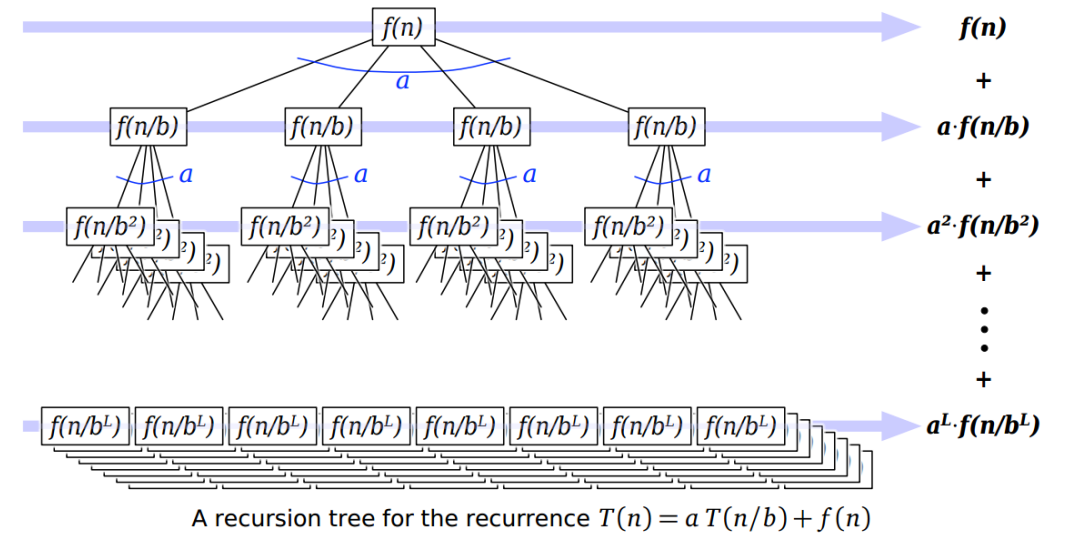
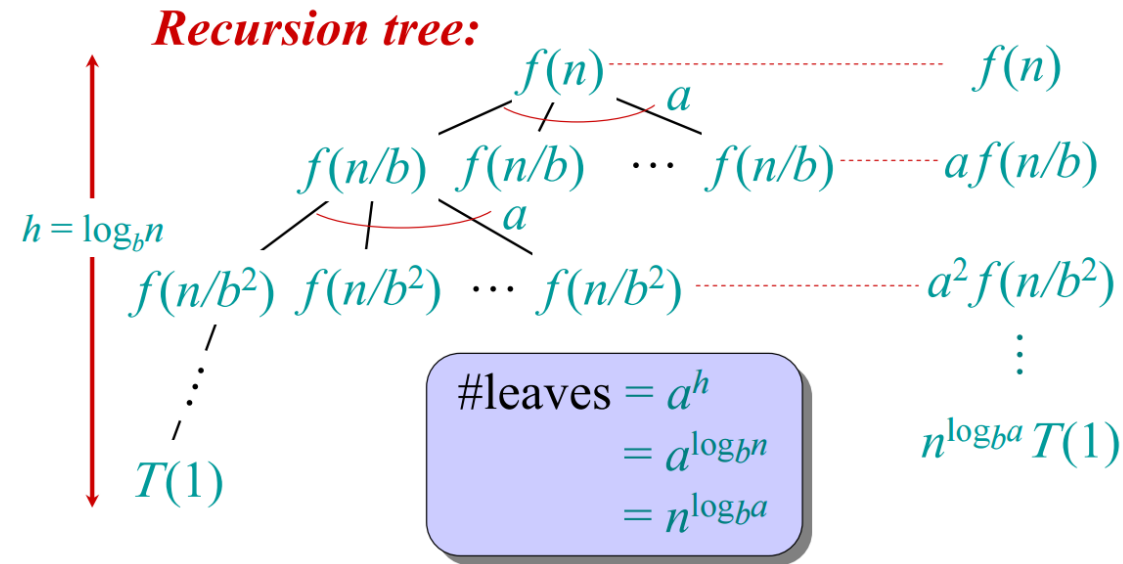
$$T(n) = a \cdot T(n/b) + f(n)$$





# Master Method: Recursion Tree

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





## Master Method: Prerequisites

Prerequisites for the application of the master method:

- $a, b$  constants  
*method not suited for variable resizing of problem*
- $a \geq 1$   
*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
- $T_2(n) = \frac{1}{2} \cdot T_2(n/3) + n/2$   
→ not applicable:  $a < 1$
- $T_3(n) = 3 \cdot T_3(n/2^{\cdot 1}) + \log(n)$   
→ 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  $n^x$ .

$\exists \varepsilon > 0: f(n) \in O(n^{x-\varepsilon}) \rightarrow$  e.g.  $\varepsilon = 0.8$

⇒ Case ①, running time dominated by the cost at the leaves

---

– Step 5: Write down result:

$T(n) \in \Theta(n^x) = \Theta(n^{\log_2(7)}) \approx \Theta(n^{2.807})$



## 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  $T(\lfloor n/b \rfloor)$  or  $T(\lceil n/b \rceil)$ .



## Substitution Method / Inductive Proof: Example

Task 1.4b from midterm 1 of FS 2016:

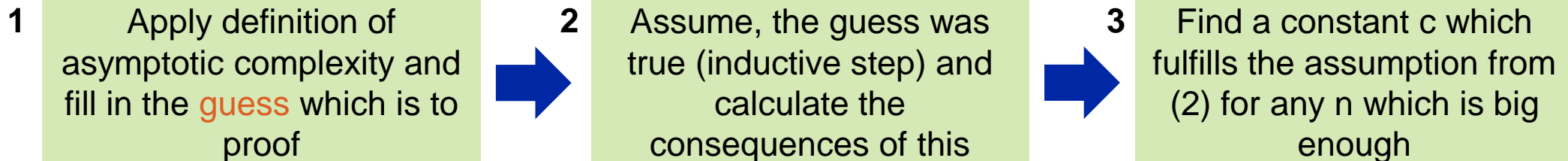
Consider the following recurrence: 
$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/5) + T(7n/10) + n & \text{if } n > 1 \end{cases}$$

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:



## 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  $g(n) = c \cdot n$  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:

$$(0.9 \cdot 10 + 1) \cdot n \leq 10 \cdot n$$

$$(9 + 1) \cdot n \leq 10 \cdot n$$

$$10 \cdot n \leq 10 \cdot n$$

...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 succeeded.

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

Therefore, we have found a witness  $c$  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 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:

$$(0.9 \cdot c + 1) \cdot n \leq c \cdot n \quad | : n$$

$$(0.9 \cdot c + 1) \leq c \quad | - 0.9 \cdot c$$

$$1 \leq c - 0.9 \cdot c \quad | \text{subtraction}$$

$$1 \leq 0.1 \cdot c \quad | \cdot 10$$

$$10 \leq c$$



## Exercise: Substitution Method / Inductive Proof

Consider the following recurrence:  $T(n) = T(n/3) + 2 \cdot T(n/9) + n$

Show that  $T(n)$  is in  $O(n)$ .



## Exercise: Substitution Method / Inductive Proof

Consider the following recurrence:  $T(n) = 2 \cdot T(n - 1) + 42$ ;  $T(1) = 1$

Show that  $T(n)$  is in  $O(2^n)$ .



## Exercise: Substitution Method / Inductive Proof

Consider the following recurrence:  $T(n) = 4 \cdot T(n / 2) + n^2$ ;  $T(1) = 1$

Show that  $T(n)$  is in  $O(n^2 \log(n))$ .



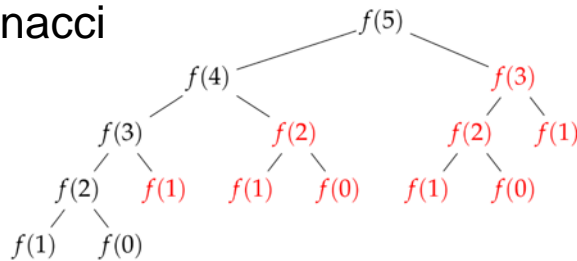


## Overview of Some Recurrences

<i>Recurrence</i>	<i>Big O solution</i>	<i>Example algorithm</i>
$T(n) = T(n/2) + O(1)$	$O(\log(n))$	Binary search
$T(n) = T(n - 1) + O(1)$	$O(n)$	Sequential search
$T(n) = 2 \cdot T(n/2) + O(1)$	$O(n)$	Tree traversal
$T(n) = T(n - 1) + O(n)$	$O(n^2)$	Selection sort
$T(n) = 2 \cdot T(n/2) + O(n)$	$O(n \log(n))$	Mergesort

## Recurrences of some Well-Known Algorithms / Problems

### – Fibonacci



```
int fib(int n) {
    if (n < 2) { return n; }
    return fib(n-1) + fib(n-2);
}
```

$$\text{fib}(n) = \begin{cases} n & \text{if } n < 2 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{else} \end{cases}$$

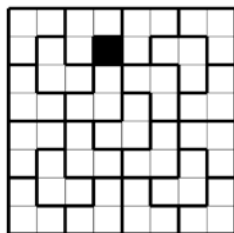
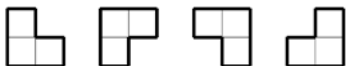
$$\rightarrow T(n) = T(n-1) + T(n-2) + O(1)$$

### – Towers of Hanoi



$$\rightarrow T(n) = 2 \cdot T(n-1) + O(1)$$

### – Tromino Tiling



$$\rightarrow T(n) = 4 \cdot T(n/2) + O(1)$$



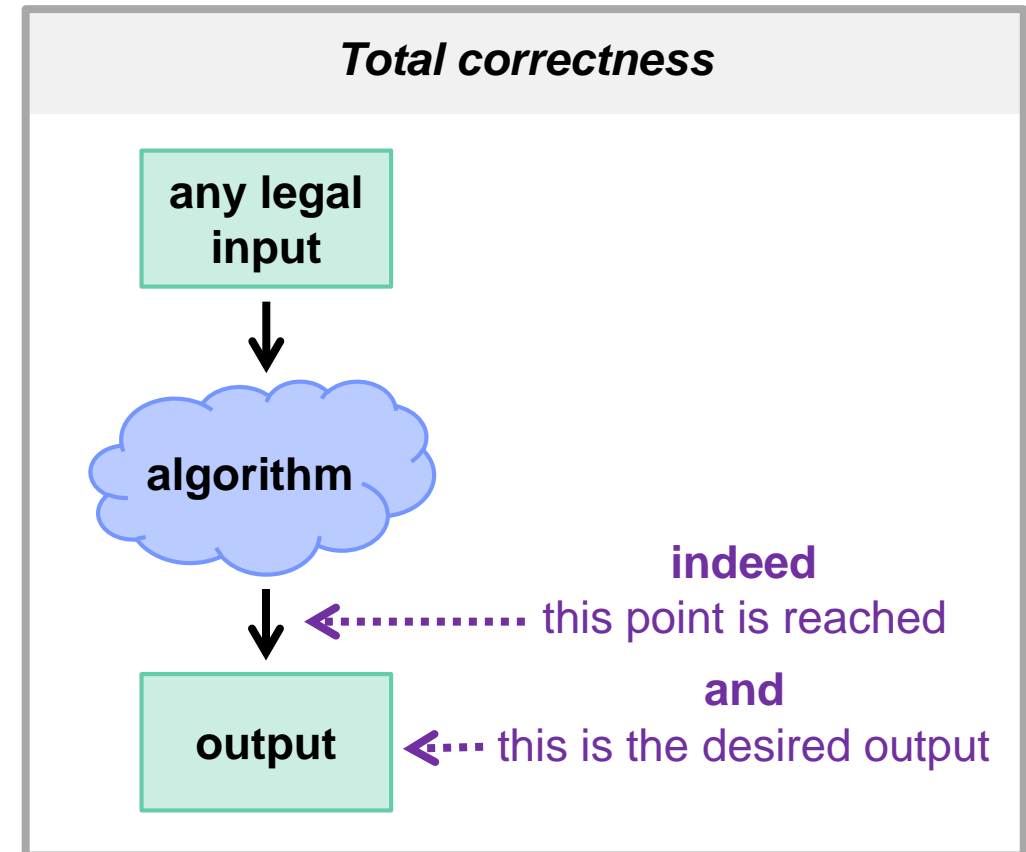
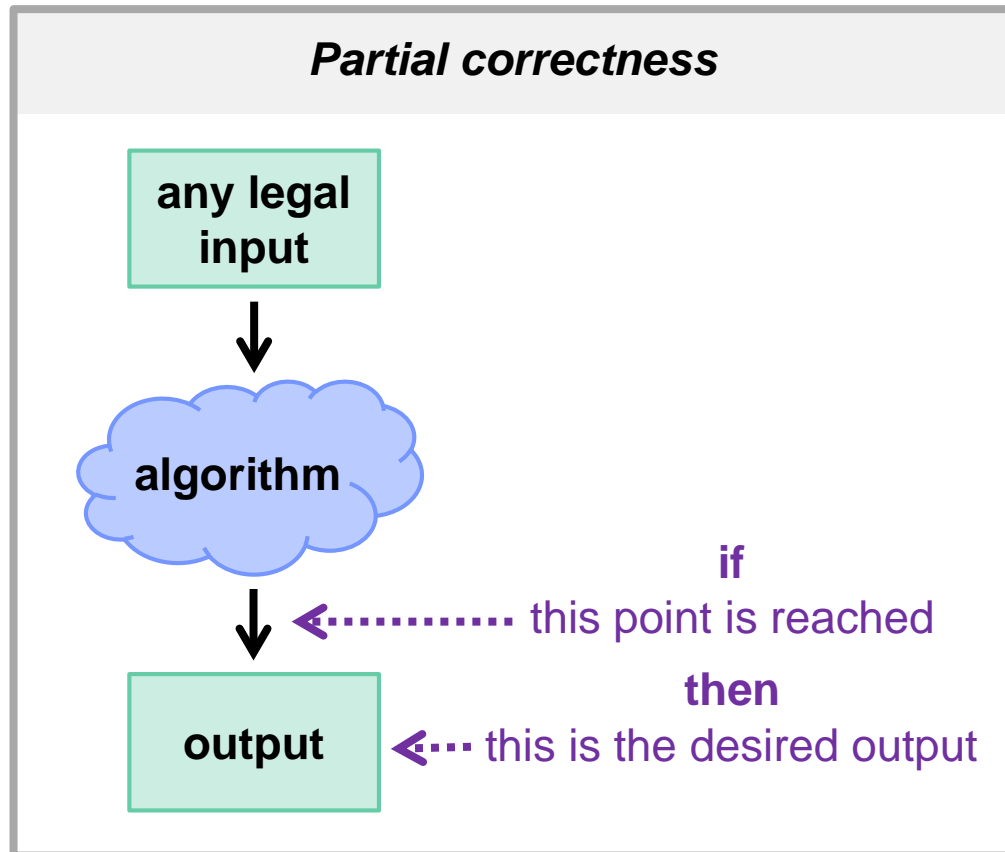
**Universität  
Zürich** UZH

**Institut für Informatik**

# Correctness, Loop Invariants

- Correctness
- Loop Invariants

## Correctness



## Correctness: Example

The following C code is a very simple example of the implementation of an algorithm which is only partially correct but not (totally) correct:

If the input is even, it will return the threefold of the input. If the input is odd, the function will go into an infinite loop and never terminate.

```
/**  
 * returns the threefold of the input  
 */  
int triple(int input) {  
    if (input % 2 == 0) {  
        return input * 3;  
    }  
}
```

## Invariants

Invariants are an **useful tool** for

- **proofing the correctness** of an algorithm and in particular of loops (loop invariants),
- better understanding and reasoning about algorithms in general.

An invariant is a **logical expression**, i.e. a statement which can be evaluated to true or false. This logical expression refers to some property of an algorithm or a part of an algorithm and **has to be true** in a certain range of the control flow for the algorithm to work as it should. Oftentimes, the invariant **captures the main idea** underlying an algorithm.

An invariant could look as follows for **example**:

- «the value of variable  $x$  has always to be positive», formally:  $x > 0$
- «all numbers from 1 to 41 are present somewhere in array  $A$  of length  $n$ »,  
formally:  $\forall x: 0 < x < 42, \exists i \leq n: A[i] = x$



## Loop Invariants

For every loop, a loop invariant can be assigned.

To show that a loop invariant holds, three conditions have to be checked:

- **initialization**: the invariant has to be true (right) before the first iteration of the loop starts
- **maintenance**: if it is true before an iteration then it is true after that iteration; the invariant has to be true right at the start and right at the end of each iteration of the loop
- **termination**: the invariant has to be true (right) after the loop has ended; this is a useful property that helps to show that the algorithm is correct

This technique is similar to an inductive proof.

## Invariant Finding Example

Find an **invariant** for the loop in the following C code fragment:

```
int c = 42;
int x = c;
int y = 0;
while (x > 0)
{
    x = x - 1;
    y = y + 1;
}
```

$c = x + y = 42$

is an invariant of the while loop in this C code fragment.

Note that this is not the only invariant of the loop but there are many possibilities, e.g.

- $y \geq 0$
- $x \geq 0$
- $(x \% 2 = y \% 2) \wedge (x \cdot y \neq 1)$
- ...

Obviously, in practice we will only be interested in an invariant which is useful for understanding the algorithm and proving a certain property of it. What is a helpful invariant, depends on what the algorithm does.



## Loop Invariants: Strategies

Two parts can be distinguished when working with loop invariants:

- finding the loop invariant,
- writing the loop invariant using formal language.

An **unavoidable prerequisite** for formulating a loop invariant is to **understand what the loop actually does** and should accomplish within the algorithm it is part of. Loop invariants are a formalization of one's intuition about how the loop works. For this part, it is difficult to give some general advice or recipes because algorithms use quite different ideas. Some algorithms use make use of similar ideas and it therefore helps if one has seen some examples of loop invariants.

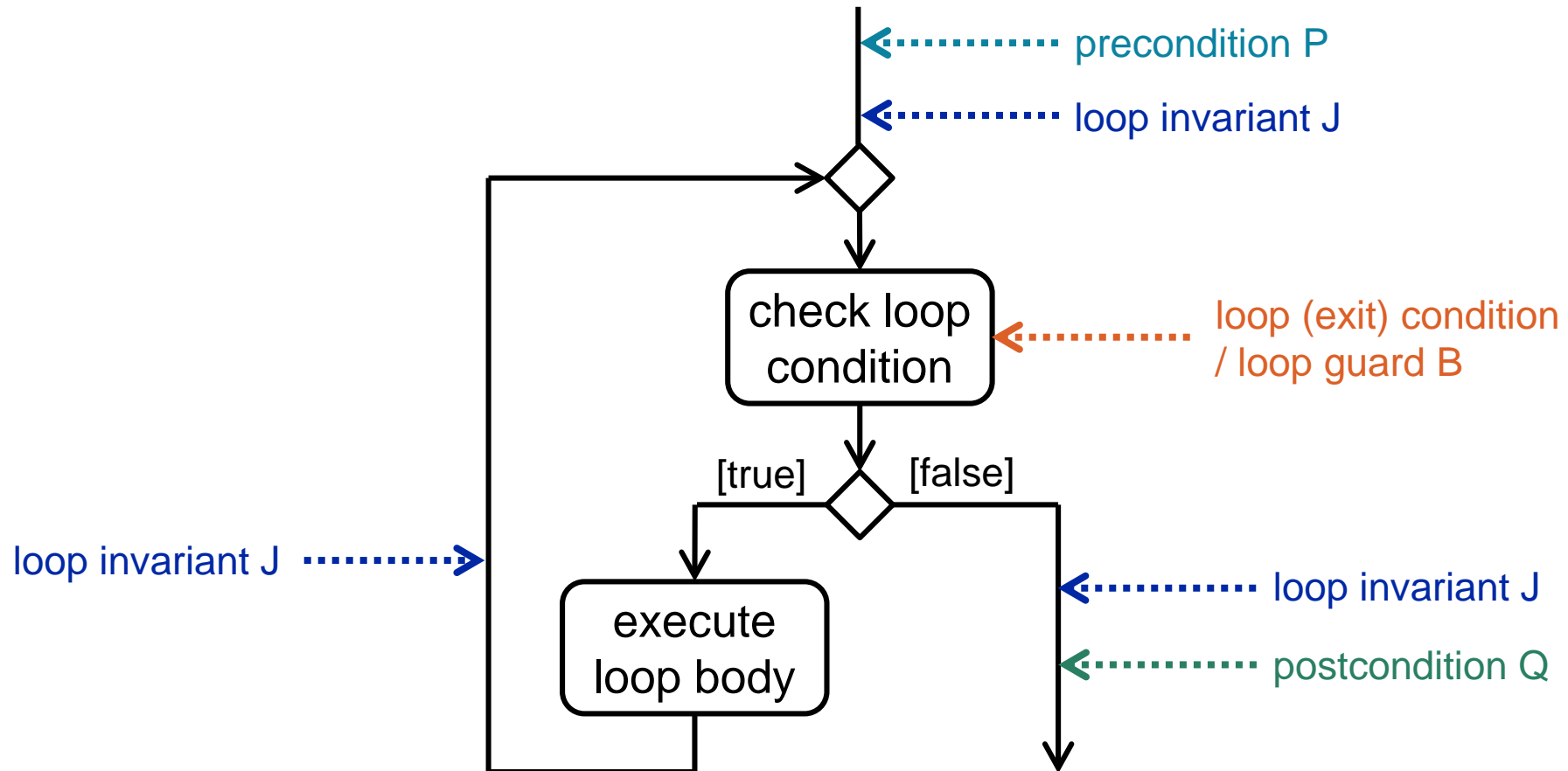
If the loop invariant was found and formulated using natural language, it has to be written using formal language (predicate logic). This part is mainly a matter of training.

## Precondition and Postcondition

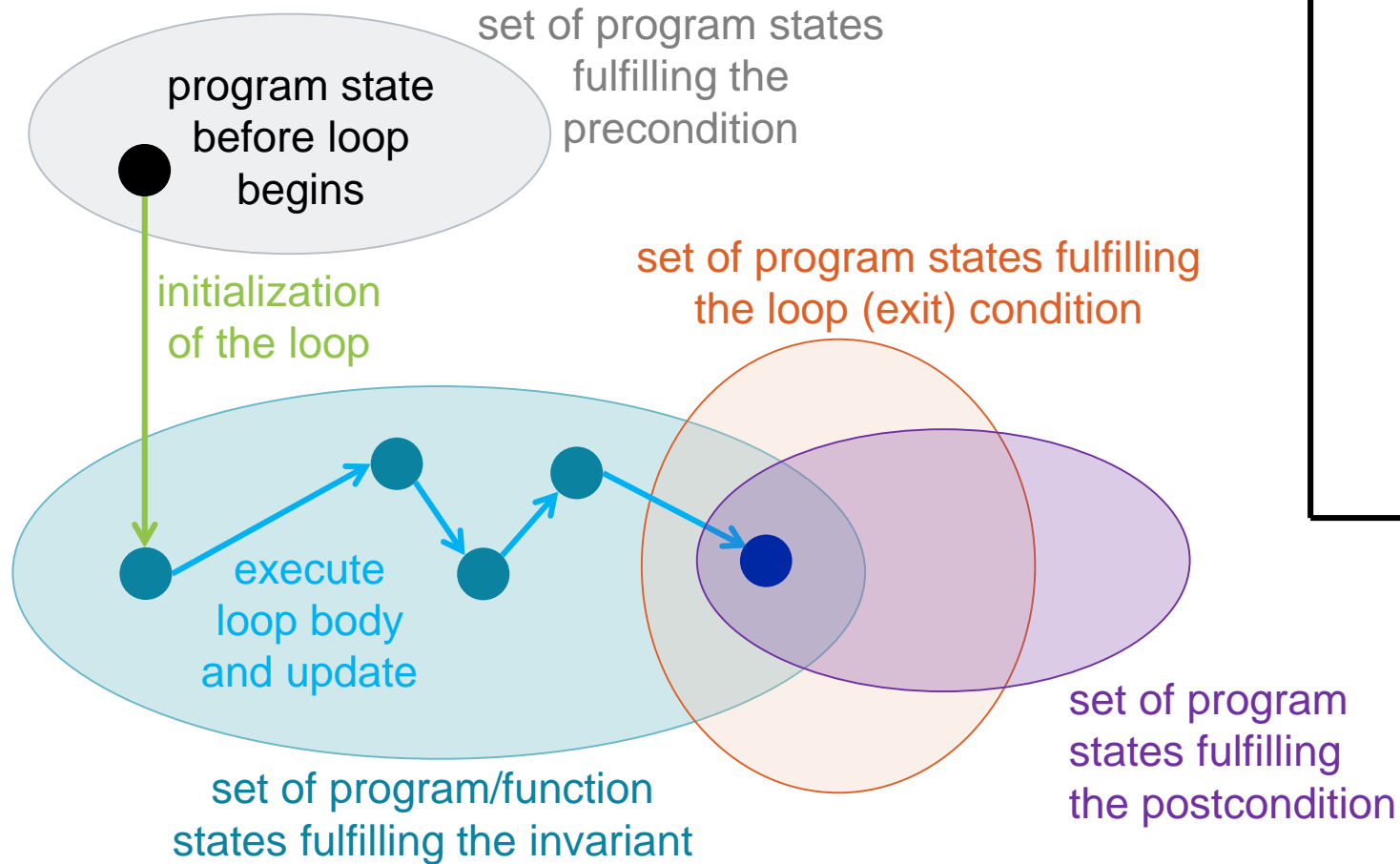
Precondition and postcondition are logical expressions which can be evaluated to true or false.

- A **precondition** defines the range / set of arguments and environment situations which allow a meaningful execution a program, function or part of it. If the input arguments do not fulfill the precondition, the result of the respective calculations are undefined. The responsibility is outsourced to the person calling the respective program function. A precondition could be for example that an input parameter might not be equal to zero (because it is used in a division).
- A **postcondition** is a description of the output or state which is reached after successful execution of a program, function or part of it.

## Loop Invariants, Precondition, Postcondition



## Loop Invariants: Visualization as States and Sets

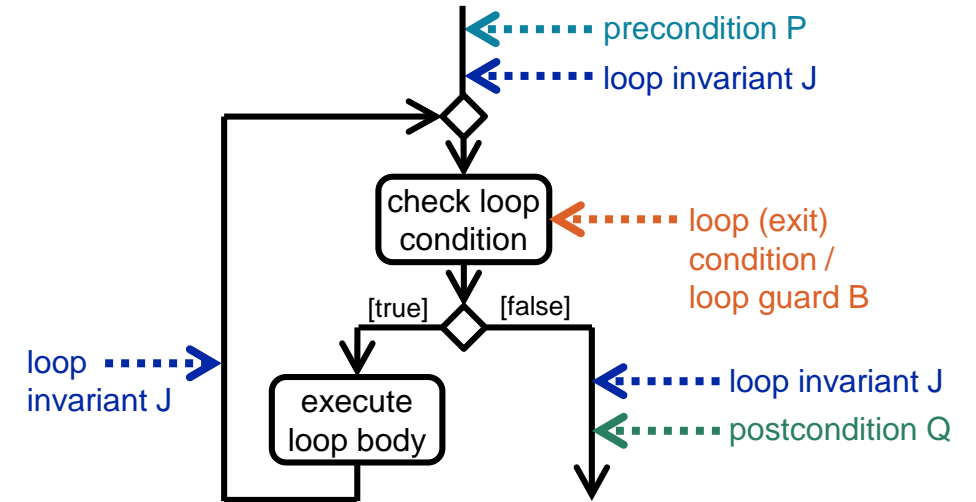


*Example:*

```
int z = 3
for (int i = 0; i < 4; i++)
{
    printf("%d"; i / z);
    //do something
}
```

## Loop Invariants: Formal Description

Using precondition  $P$ , postcondition  $Q$ , loop exit condition  $B$  and loop invariant  $J$ , we can now proceed to a more formal description of the terms initialization, maintenance and termination:



- **Initialization**: Invariant must hold initially:  $P \Rightarrow J$
- **Maintenance**: loop body must re-establish the invariant:  $(J \wedge B) \Rightarrow J$
- **Termination**: Invariant must establish postcondition if loop condition is false:  $(J \wedge \neg B) \Rightarrow Q$

## Postconditions of Sorting Algorithms

Any sorting algorithm, regardless of how it works, has to fulfill the following **two postconditions** to achieve partial correctness:

- The array **elements must be sorted**:  $A[1..n]$  is sorted
  - more formally:  $\forall i ((i > 0 \wedge i < n-1) \rightarrow (A[i] \leq A[i+1]))$
- The array **elements may not have been changed**:  $A[1..n] \in A^{\text{orig}}$ 
  - another way to state this: the output array is a permutation ( $\pi$ ) of the input array:  $A_{\text{out}} = \pi(A_{\text{in}})$
- The second of the two conditions is forgotten relatively easily but is crucial. Without this condition one could write an algorithm which just always returns the array  $[1,2,3,4]$  no matter what the input is. This would fulfill the first condition but obviously is not what is expected from a sorting algorithm.



## Expressing Statements in Predicate Logic: Exercise

When dealing with correctness and loop invariants, one difficulty is to be able to precisely express statements about an algorithm mathematically (using predicate logic with quantors  $\exists$  and  $\forall$  and logical operators  $\wedge$  and  $\vee$ ).

Let  $A[1..n]$  be an array of  $n$  integers.

Express the following statements as formulas in predicate logic:

- a) All elements of  $A$  are greater than 42.
- b) All elements with array index greater than 5 are odd.
- c) There are no two elements with identical values in index range from  $a$  (inclusive) to  $b$  (exclusive).
- d) The elements with array index smaller than  $m$  are sorted descendingly.



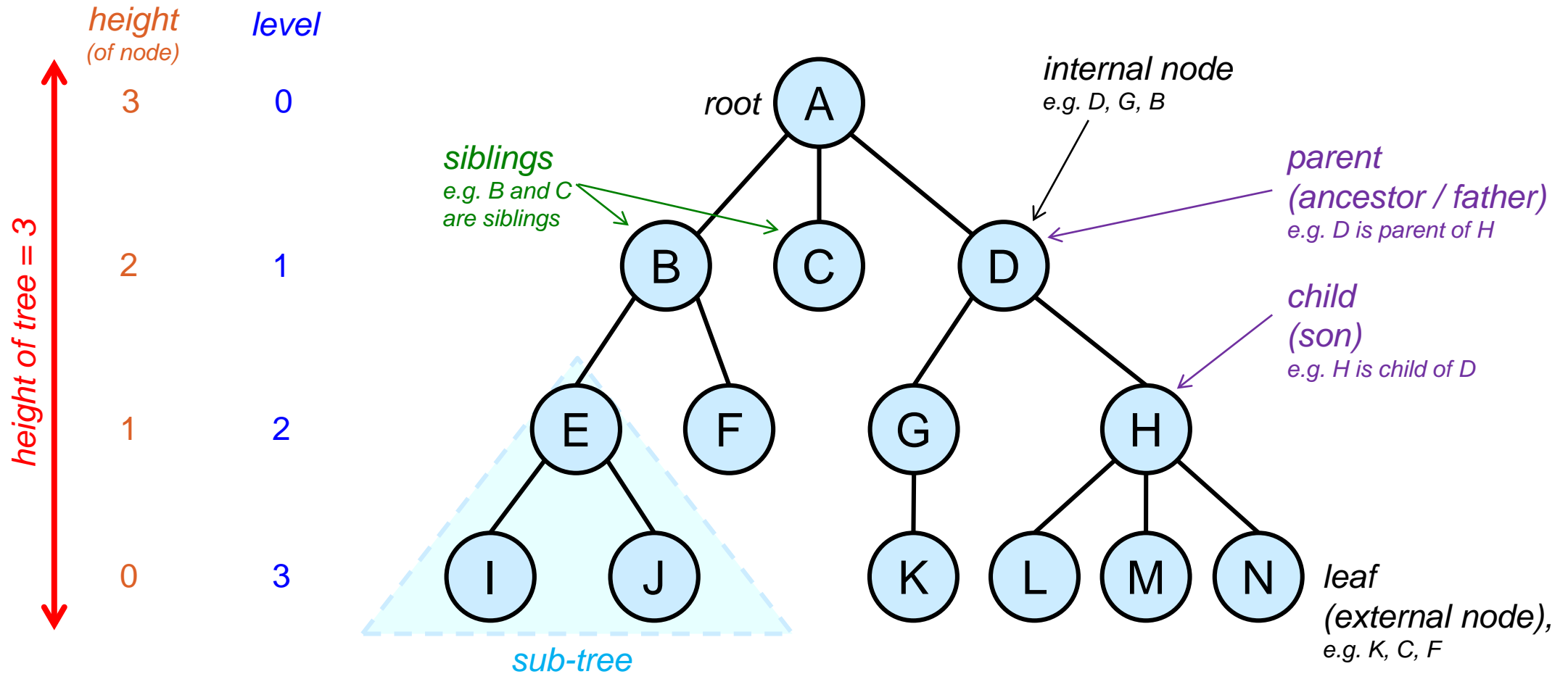
# Trees

- Nomenclature





## Trees: Nomenclature





# Heaps

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

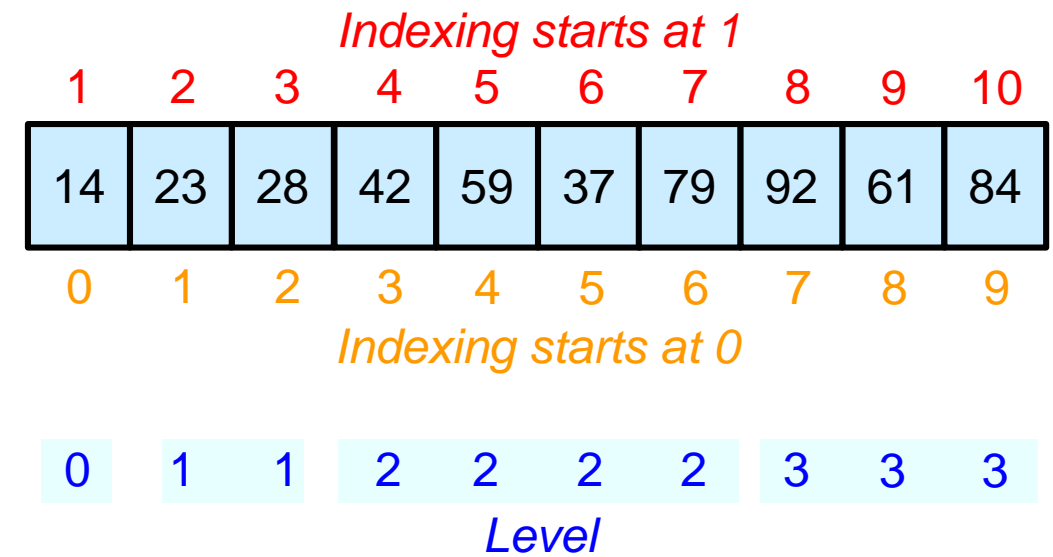
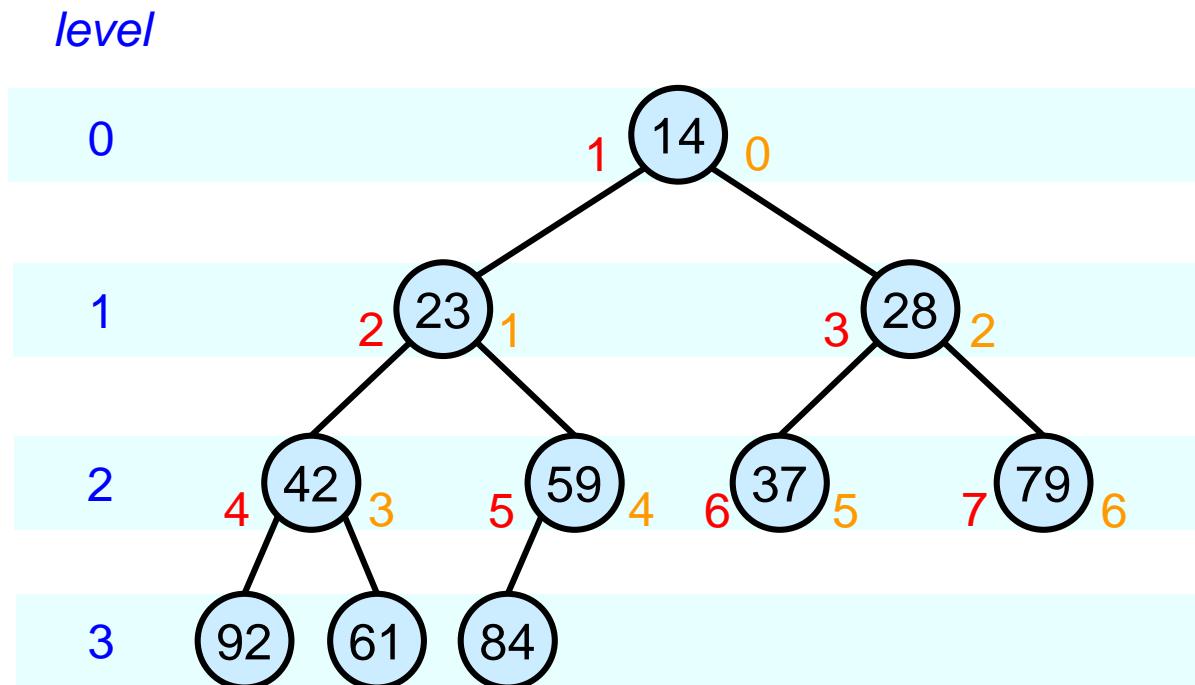




## Heap Conditions

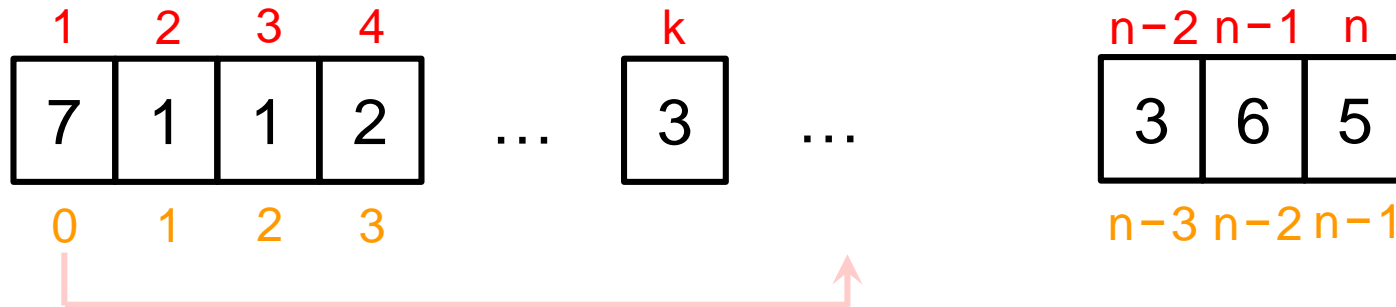
- A heap is a **binary tree**.
- All **levels except** the **lowest** are **fully filled** and the lowest level is occupied starting from left without gaps (= «nearly complete tree»).
- Keys in nodes of tree are ordered:
  - in a **max-heap**: for all nodes, the key is bigger than or equal to all its 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

## Representation of a Heap as an Array



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

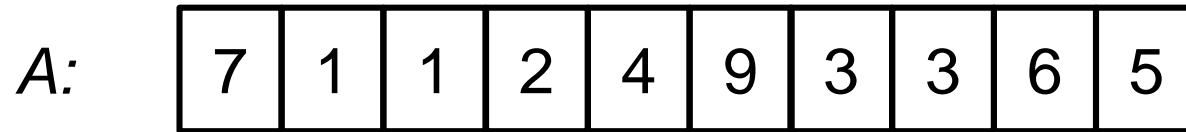
## Addressing Parent and Child Nodes



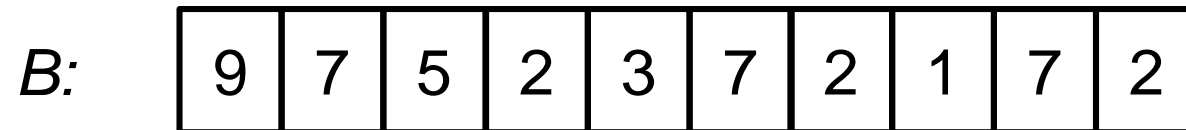
	Indexing starts at 1	Indexing starts at 0
Getting to <i>parent</i> of node with index $k$	$\lfloor k / 2 \rfloor$	$\lfloor (k - 1) / 2 \rfloor$
Getting to <i>left child</i> of node with index $k$	$2 \cdot k$	$2 \cdot k + 1$
Getting to <i>right child</i> of node with index $k$	$2 \cdot k + 1$	$2 \cdot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

## Heap Conditions: Exercise

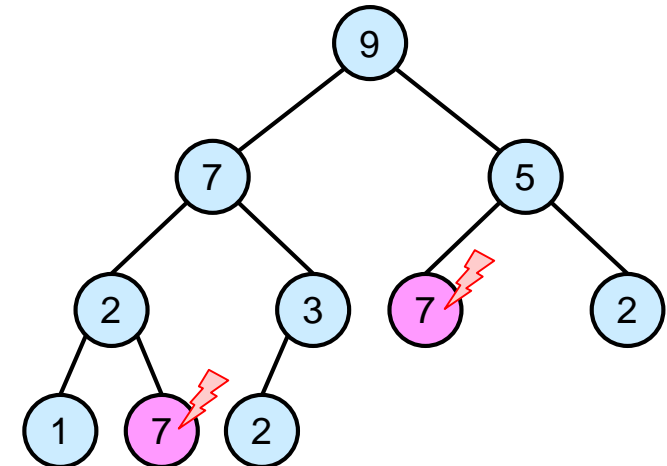
*The following arrays shall each represent a binary heap. Do they fulfill the conditions of a max-heap?*



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



→ 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.





## 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.

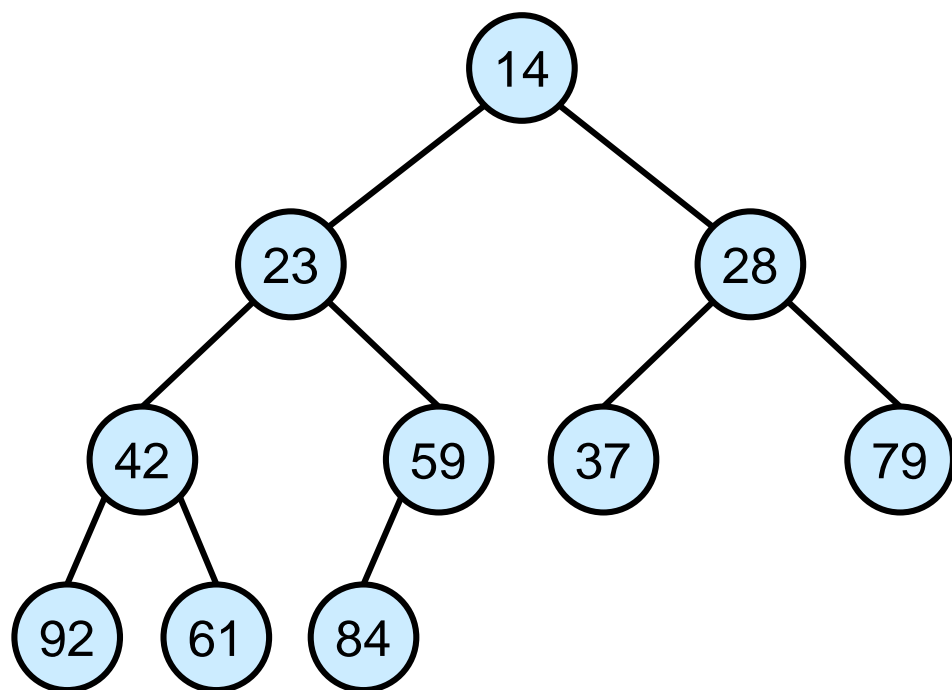


## Heaps: Sorting Algorithm

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

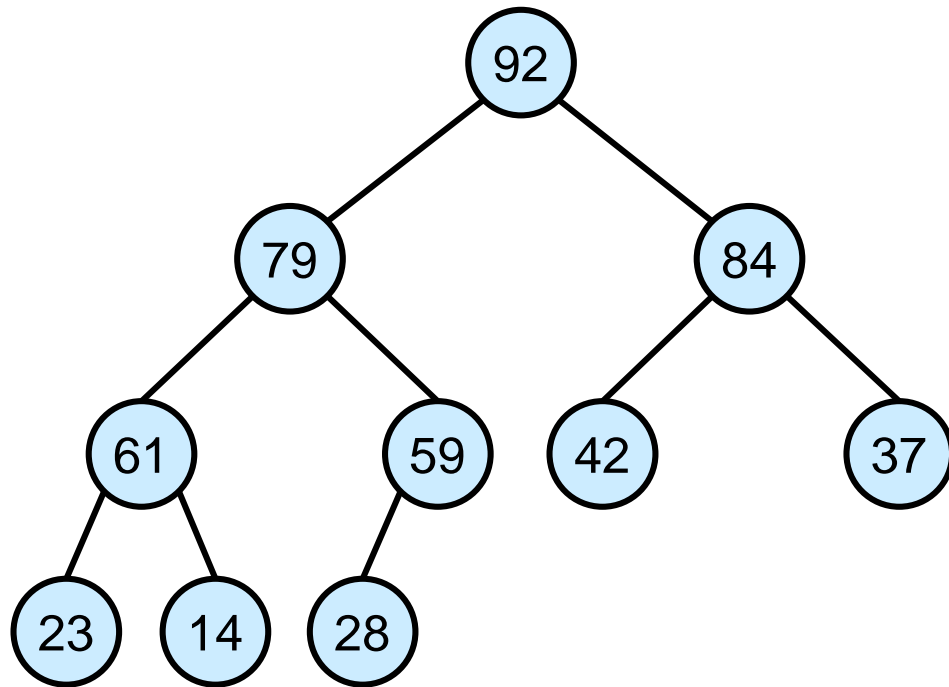


## Heaps: Demonstration: Removing the Root

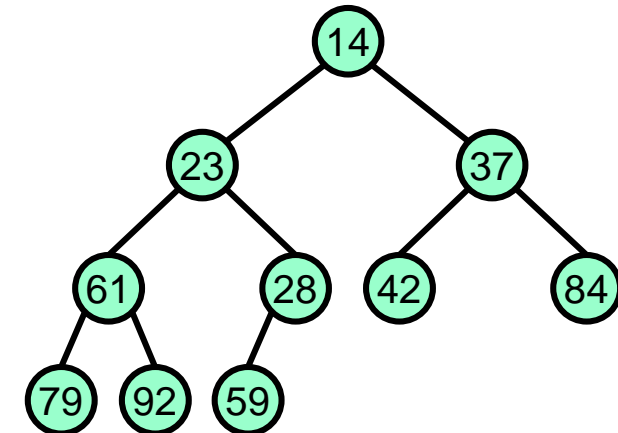


14	23	28	42	59	37	79	92	61	84
23	42	28	61	59	37	79	92	84	14
28	42	37	61	59	84	79	92	23	14
37	42	79	61	59	84	92	28	23	14
42	59	79	61	92	84	37	28	23	14
59	61	79	84	92	42	37	28	23	14
61	84	79	92	59	42	37	28	23	14
79	84	92	61	59	42	37	28	23	14
84	92	79	61	59	42	37	28	23	14
92	84	79	61	59	42	37	28	23	14
92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

## Heaps: 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: Comprehension Question

Why do we iterate from  $\lfloor n/2 \rfloor$  to 1 and not from 1 to  $\lfloor n/2 \rfloor$  in the BuildHeap algorithm?  
(Does this even matter?)

**Algo:** Heapify(A,i,s)

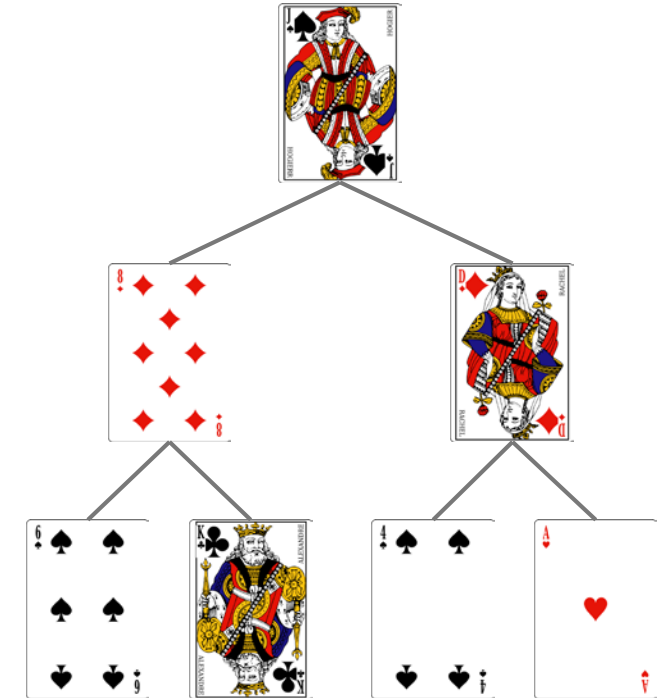
```
m = i;
l = Left(i);
r = Right(i);
if  $l \leq s \wedge A[l] > A[m]$  then m = l;
if  $r \leq s \wedge A[r] > A[m]$  then m = r;
if  $i \neq m$  then
    exchange A[i] and A[m];
    Heapify(A,m,s);
```

**Algo:** BuildHeap(A)

```
for  $i = \lfloor n/2 \rfloor$  to 1 do Heapify(A,i,n);
```

## 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 tree 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.





**Universität  
Zürich** <sup>UZH</sup>

**Institut für Informatik**

# Preview on Exercise 5



## Exercise 5 – Tasks 1 and 2: Heap and Heapsort

Look at the respective slides from the lecture: SL04, pp. 5 – 33.



## Exercise 5 – Task 3: Quicksort

Look at the respective slides from the lecture: SL04, pp. 35 – 53.



**Universität  
Zürich** <sup>UZH</sup>

**Institut für Informatik**

# Wrap-Up

- Summary
- Outlook





*Thank you for your attention.*