



Universität
Zürich ^{UZH}

Institut für Informatik

Informatics II

Tutorial Session 5

Wednesday, 23rd of March 2022

Discussion of Exercises 3 and 4,
Divide and Conquer, Recurrences

14.00 – 15.45

BIN 0.B.06



Agenda

- Review of Exercise 3 (continued)
- Review of Exercise 4
 - Recurrences: Repeated Substitution, Recursion Tree, Master Method (Tasks 2 to 5)
 - Application of the Divide and Conquer Technique (Tasks 6 and 7)
- (Preview on Exercise 5)



Review of Exercise 3 (continued)

- Task 1



Exercise 3, Task 1

Algo: whatDoesItDo(A, n, k)

result = -1000

for $i = 1$ **to** n **do**

current = 0

for $j = i$ **to** n **by** k **do**

current = *current* + $A[j]$

if $current > result$ **then**

result = *current*

return *result*

Tips and Tricks for Finding Out What an Algorithm Does

- Having experience (i.e. having seen, thought about and understood many algorithms) and practicing definitely helps.
- Looking at the variable names may help (assuming they are not intentionally misleading).
- Find out what parts of the code are intended for (i.e. lines 8 to 10 are a swap) and wrap them into an abstraction (i.e. write «swap $A[i]$ and $A[\max]$ » instead of lines 8 to 10).
- Make a drawing of the array on which the algorithm operates and think about how the loops go over it.
- Make a table and trace how the variables and their values change.
- Calculate a set of input and output values and compare them.
- ...

↓ increasing desperation

Exercise 3, Task 1a and 1b

- a) Assume $A[1..n] = [1, 3, 5, 4, 2, 6, 8]$ and $k = 3$. Describe how the function works step by step and calculate what should be returned by this function.
- b) Describe what the algorithm does and what the output is in general cases.

$A =$

1	3	5	4	2	6	8
1	2	3	4	5	6	7

Algo: whatDoesItDo(A, n, k)

```
result = -1000
for i = 1 to n do
    current = 0
    for j = i to n by k do
        current = current + A[j]
    if current > result then
        result = current
return result
```

Exercise 3, Task 1a and 1b

Alternative solution: use a trace table

Algo: whatDoesItDo(A, n, k)

```
result = -1000
for  $i = 1$  to  $n$  do
  current = 0
  for  $j = i$  to  $n$  by  $k$  do
     $\lfloor$  current = current +  $A[j]$ 
    if current > result then
       $\lfloor$  result = current
return result
```


Running Time Analysis: Loop With Arbitrary Fixed Step Size

In case of a (well-behaved) for loop with an **arbitrary** (but constant and integer) **step size**, the number of executions of the body can be calculated as follows:

up-counting:
$$\left\lfloor \frac{\text{end} - \text{start} + \text{step size}}{\text{step size}} \right\rfloor$$

down-counting:
$$\left\lfloor \frac{\text{start} - \text{end} + \text{step size}}{\text{step size}} \right\rfloor$$

Examples:

<i>START</i>	<i>END</i>	<i>STEP</i>	<i>Number of times executed</i>
for <i>i</i> = 5 to <i>n</i> step 5 do			$\lfloor n / 5 \rfloor + 1$
[<loop body>			$\left\lfloor \frac{\text{END} - \text{START} + \text{STEP}}{\text{STEP}} \right\rfloor = \lfloor n / 5 \rfloor$

<i>START</i>	<i>END</i>	<i>STEP</i>	<i>Number of times executed</i>
for (<i>i</i> = 57; <i>i</i> >= 23; <i>i</i> = <i>i</i> - 3) {			13
/* loop body */			12
}			

Running Time Analysis: Overview of Formulas for Calculation of Number of Executions

<i>counting direction</i>	<i>boundary inclusive?</i>	<i>step size</i>	<i>code example</i>	<i>no. of executions of the loop body</i>	<i>no. of executions of the loop header</i>
up	yes	1	<code>for(i = 1; i <= 99; i++)</code>	end – start + 1	end – start + 2
down	yes	1	<code>for(i = 99; i >= 1; i--)</code>	start – end + 1	start – end + 2
up	no	1	<code>for(i = 1; i < 99; i++)</code>	end – start	end – start + 1
down	no	1	<code>for(i = 99; i > 1; i--)</code>	start – end	start – end + 1
up	yes	arbitrary	<code>for(i = 1; i <= 99; i=i+3)</code>	$\left\lfloor \frac{\text{end} - \text{start} + \text{step}}{\text{step}} \right\rfloor$	$\left\lfloor \frac{\text{end} - \text{start} + \text{step}}{\text{step}} \right\rfloor + 1$
down	yes	arbitrary	<code>for(i = 99; i >= 1; i=i-7)</code>	$\left\lfloor \frac{\text{start} - \text{end} + \text{step}}{\text{step}} \right\rfloor$	$\left\lfloor \frac{\text{start} - \text{end} + \text{step}}{\text{step}} \right\rfloor + 1$

Exercise 3, Task 1c and 1d

- c) *Perform an exact analysis of the running time of the algorithm.*
- d) *Determine the asymptotic complexity of the algorithm.*

Algo: whatDoesItDo(A, n, k)

```
result = -1000
for i = 1 to n do
    current = 0
    for j = i to n by k do
        current = current + A[j]
    if current > result then
        result = current
return result
```

Exercise 3, Task 1c and 1d

Instruction	# of times executed	Cost
result = -1000	1	c_1
for $i := 1$ to n do	$n + 1$	c_2
$current = 0$	n	c_3
for $j := i$ to n by k do	$\frac{n^2 - n}{2k} + 2n^*$	c_4
$current = current + A[j]$	$\frac{n^2 - n}{2k} + n^{**}$	c_5
if $current > result$ then	n	c_6
$result = current$	αn^{***}	c_7
return $result$	1	c_8

Exercise 3, Task 1e

```
void maximal_sum_every_k(int A[], int n, int k) {  
    int result = -1000;  
    int i;  
    int j;  
    for (i = 0; i < n; i++) {  
        int current = 0;  
        for (j = i; j < n; j += k) {  
            current = current + A[j];  
        }  
        if (current > result) {  
            result = current;  
        }  
    }  
    printf("Result: %d\n", result);  
}
```

Algo: whatDoesItDo(A, n, k)

```
result = -1000  
for  $i = 1$  to  $n$  do  
     $current = 0$   
    for  $j = i$  to  $n$  by  $k$  do  
         $current = current + A[j]$   
    if  $current > result$  then  
         $result = current$   
return result
```



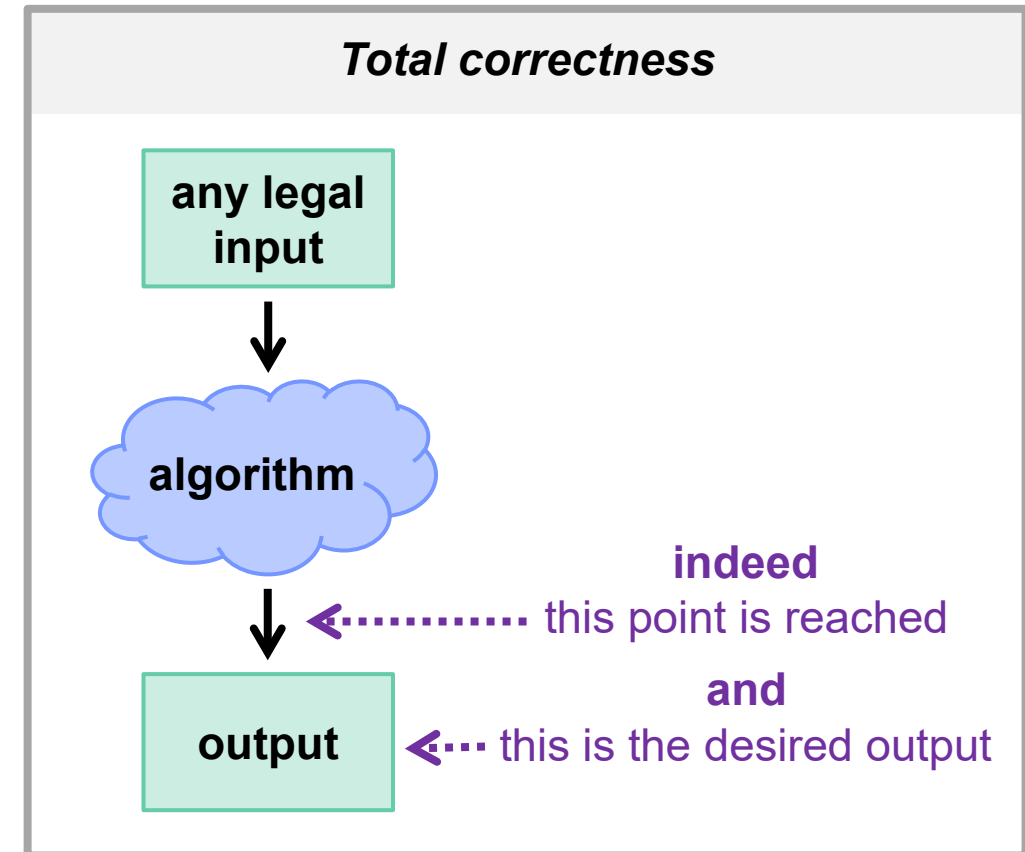
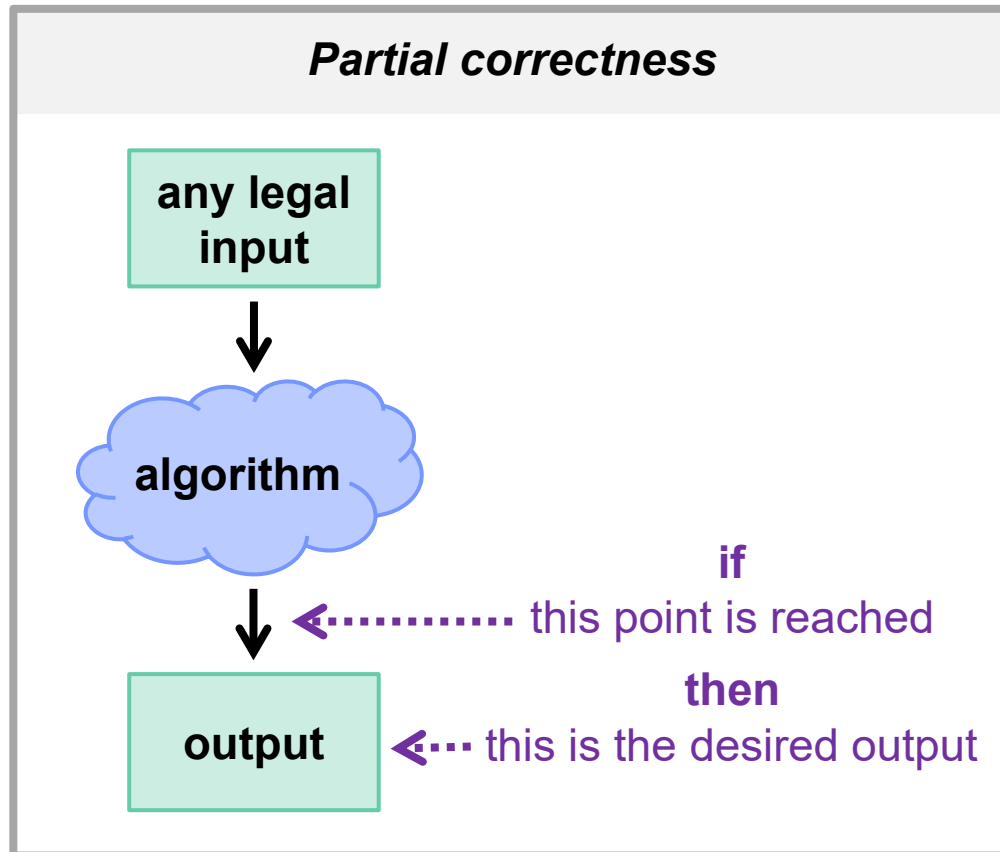
**Universität
Zürich** UZH

Institut für Informatik

Correctness, Loop Invariants

- Correctness
- Loop Invariants

Correctness



Correctness: Example

The adjacent 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.

```
1  /**
2   * returns the threefold of the input
3   */
4  int triple(int input) {
5      if (input % 2 == 0) {
6          return input * 3;
7      }
8      else {
9          while (1 == 1) {
10             input = input + 1;
11         }
12     }
13 }
```


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.

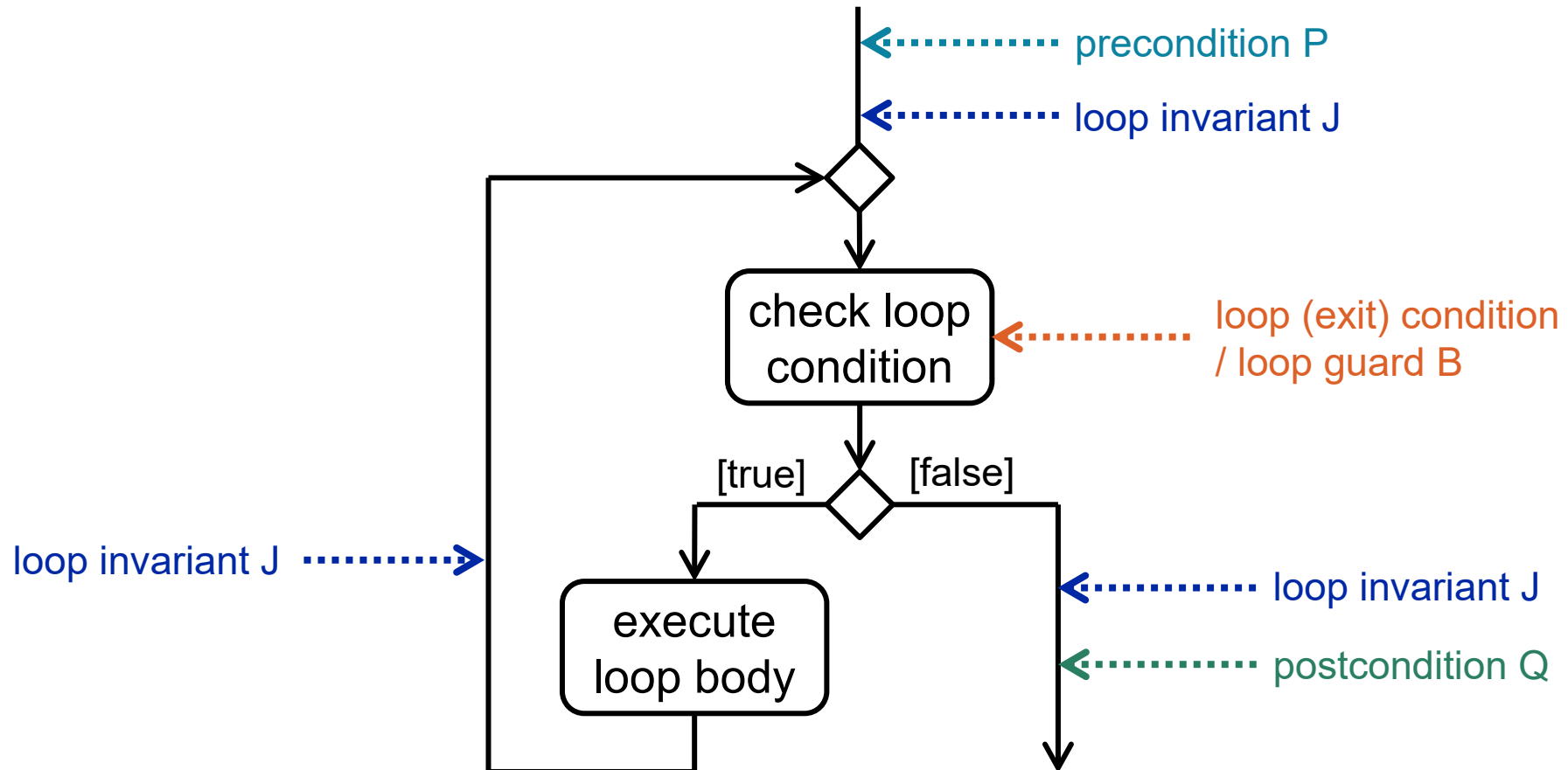
If the loop invariant was found and formulated using natural language, it has to be written using formal language (predicate logic). This part is mainly a matter of training.

Precondition and Postcondition

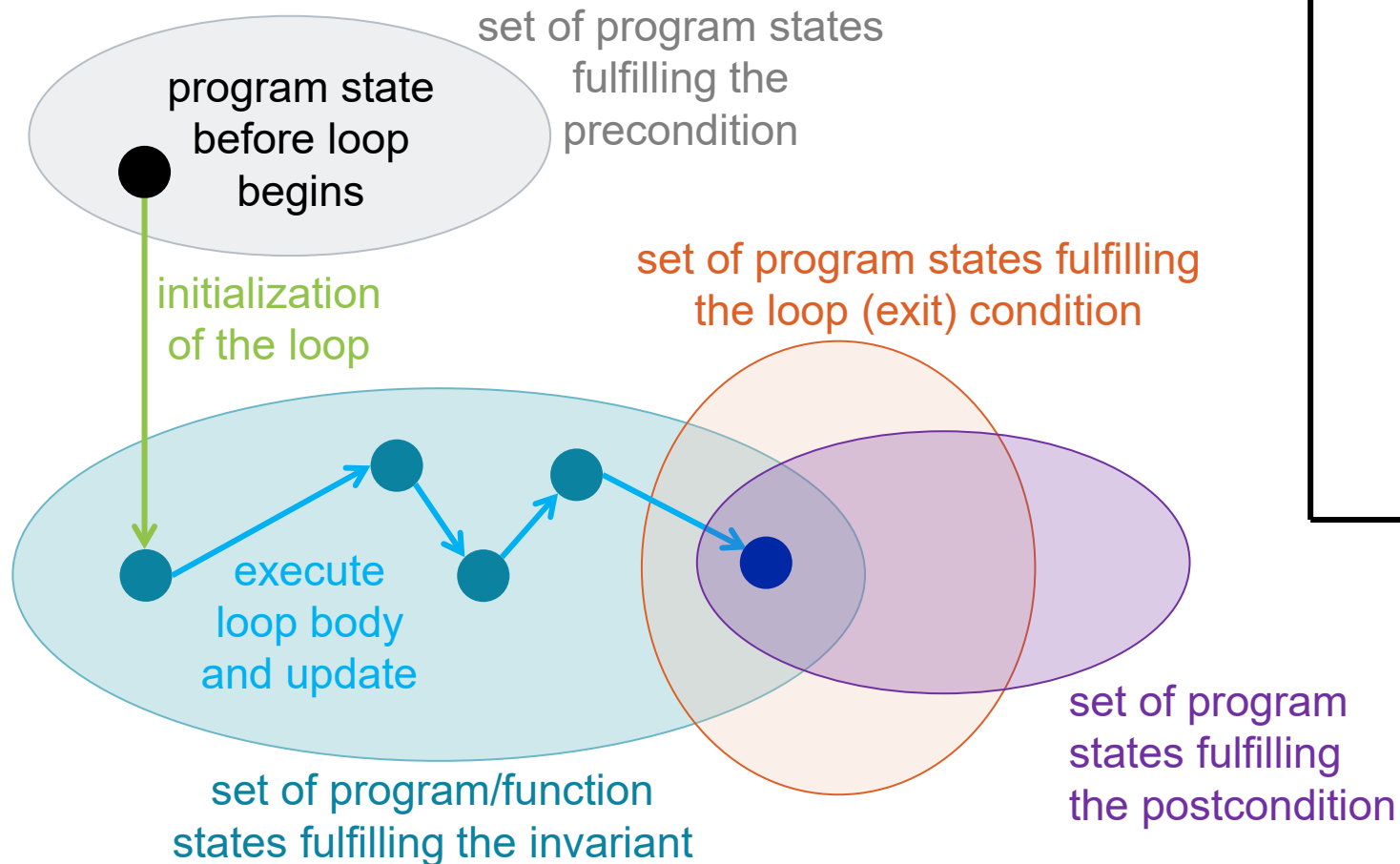
Precondition and postcondition are logical expressions which can be evaluated to true or false.

- 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

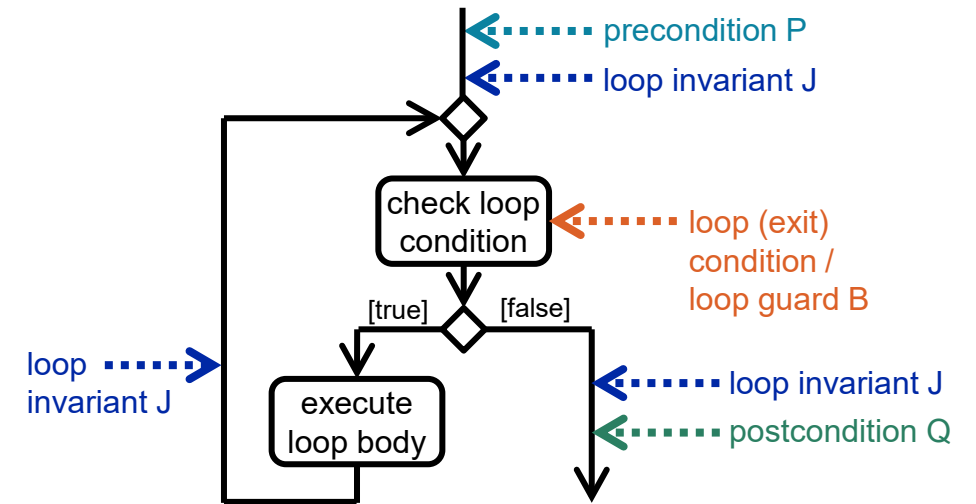


Example:

```
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: $(J \wedge \neg B) \Rightarrow Q$



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.

Exercise 3, Task 3

Algo: algo1(A, n, k)

```
sum = 0;
for  $i = 1$  to  $k$  do
     $maxi = i$ ;
    for  $j = i$  to  $n$  do
        if  $A[j] > A[maxi]$  then
             $maxi = j$ ;
     $sum = sum + A[maxi]$ ;
     $swp = A[i]$ ;
     $A[i] = A[maxi]$ ;
     $A[maxi] = swp$ ;
return  $sum$ 
```

Exercise 3, Task 3a: Solution

a) *Specify the pre and post conditions of the algorithm.*

The preconditions (inputs) are an array $A[1..n]$ and an integer k .

The post conditions(outputs) are the following:

- sum of the biggest k elements of the array $A[1..n]$. Recursively, we can define the output of **algo1** (sum of the biggest k elements of the array $A[1..n]$) in the following way: Let $sum \in \mathbb{N}$ denote the biggest k elements of the array $A[1..n]$, then we have

$\forall i \in [1..k] : sum = sum + A[i]$, where $A[1..k]$ are the biggest k integers and sorted in a descending order

- Integers of $A[1..k]$ are the biggest k integers in A and sorted in a descending order.

Exercise 3, Task 3b: Solution

- i. Determine if the loop is up loop or down loop.

The outer loop `for i = 1 to k` is an up loop, as it runs from low (1) to high k .

The inner loop `for j = i to n` is an up loop as well, as it runs from low (i) to high n .

- ii. Determine the invariants of these two loops and verify whether they hold in three stages: **initialization**, **maintenance** and **termination**.

We start with the inner loop. The invariant of the inner loop is that $\forall k \in [i..j] : A[*maxi*] \geq A[k]$, i.e., $A[*maxi*]$ is the largest element in the array $A[i..j]$.

- i. **Initialization.** This invariant holds. At this time, $j = i$ and $A[i, j]$ contains only one element $A[i]$. At this moment (before the loop starts), we have $j = i$ and $A[*maxi*]$ is the only, and the biggest element in $A[i, j]$.
- ii. **Maintenance.** This invariant holds. If $A[*maxi*] \geq A[m], \forall m \in [i, j]$ (before the loop), and $A[j] > A[*maxi*]$, then $maxi$ is assigned to be j . In this case, we still have $A[*maxi*] \geq A[m], \forall m \in [i..j]$ (after the loop).
- iii. **Termination.** The inner loop terminates when $j = n$. At this time, if $A[j] = A[i] > A[*maxi*]$, then $maxi$ is assigned to be n . It allows us to conclude that $A[*maxi*] > A[m], \forall m \in [i..j]$. Furthermore, we also have $A[*maxi*] > A[m], \forall m \in [i..n]$ after the loop terminates. In other words, $A[*maxi*]$ is the biggest element in $A[i..n]$.

Exercise 3, Task 3b: Solution

Then we analyse the outer loop, **for** $i = 1$ **to** k . The invariant of the outer loop is that $A[1..i]$ is sorted in descending order and contains the biggest i elements of the array $A[1..n]$.

- i. **Initialization.** This invariant holds. Before the loop begins, $i = 1$ and $A[1..i]$ contains only one element. It is naturally sorted and has the largest element of the subarray $A[1..1]$.
- ii. **Maintenance.** This invariant holds. We assume $A[1..i-1]$ is sorted in descending order and contains the largest $i - 1$ elements of the array $A[1..n]$ before the loop. The inner loop guarantees that max_i is the index to the biggest element in $A[i..n]$. Since $A[1..i-1]$ already contains the biggest $i - 1$ elements of $A[1..n]$, we have

$$\forall p \in [i..n], \forall q \in [1..i-1], A[q] \geq A[p]$$

In other words, max_i is the index to the biggest element in $A[i..n]$ but it is still smaller than any element in $A[1..i-1]$, which implies it is the i th biggest element. The body of the outer loop swaps it with the i th element in $A[1..n]$, which keeps $A[1..i]$ the biggest i elements of $A[1..n]$ and sorted in descending order.

- iii. **Termination.** The loop terminates when $i = k$. At this time, $A[1..k]$ is sorted in descending order and contains the largest k elements of $A[1..n]$.

Exercise 3, Task 3c: Solution

Identify some edge cases of the algorithm and verify if the algorithm has the correct output.

- If $A[1..n]$ is empty, then the algorithm only initialize `sum` to be zero and returns it.
- If $A[1..n]$ only contains one element, the outer loop will be executed only once and guarantees the $A[1..n]$ contains the biggest element, which is the only element in the array. The algorithm returns the initialized sum (0) plus the only element in the array ($A[1]$).
- For a general case, the outer loop guarantees that $A[1..n]$ contains the biggest k elements. In the body of the outer loop, the algorithm calculates the sum of the first k elements in the array $A[1..n]$ and returns it. The returned value is the sum of the biggest k elements.

Exercise 3, Task 3d: Solution

Instruction	# of times executed	Cost
<i>sum</i> := 0	1	c_1
for <i>i</i> := 1 to <i>k</i> do	$k + 1$	c_2
<i>maxi</i> := <i>i</i>	k	c_3
for <i>j</i> := <i>i</i> to <i>n</i> do	$\left(kn - \frac{k(k+1)}{2}\right)^* + k^{**}$	c_4
if $A[j] > A[\textit{maxi}]$ then	$kn - \frac{k(k+1)}{2}$	c_5
<i>maxi</i> := <i>j</i>	$\alpha \left(kn - \frac{k(k+1)}{2}\right)^{***}$	c_6
<i>sum</i> := <i>sum</i> + $A[\textit{maxi}]$	k	c_7
<i>swp</i> := $A[i]$	k	c_8
$A[i]$:= $A[\textit{maxi}]$	k	c_9
$A[\textit{maxi}]$:= <i>swp</i>	k	c_{10}
return <i>sum</i>	1	c_{11}

Exercise 3, Task 3e: Solution

Best case

In the best case, the array has already been sorted in descending order, hence we do not need to run `maxi := j`, i.e., $\alpha = 0$. In this case,

$$T_{\text{best}}(n) = c_1 + c_2(k+2) + c_3k + c_4\left((k+1)n - \frac{k(k+1)}{2} + k\right) + c_5\left((k+1)n - \frac{k(k+1)}{2}\right) + 0 + (c_7 + c_8 + c_9 + c_{10})k + c_{11}$$

$$T_{\text{best}}(n) = O(k * n)$$

Worst case

Similarly, in the worst case, the array is sorted in ascending order and we have to update `maxi` every time, i.e., $\alpha = 1$. In this case,

$$T_{\text{worst}}(n) = c_1 + c_2(k+2) + c_3k + c_4\left((k+1)n - \frac{k(k+1)}{2} + k\right) + c_5\left((k+1)n - \frac{k(k+1)}{2}\right) + c_6\left((k+1)n - \frac{k(k+1)}{2}\right) + (c_7 + c_8 + c_9 + c_{10})k + c_{11}$$

$$T_{\text{worst}}(n) = O(k * n)$$

Asymptotic complexity of best and worst case

$$T_{\text{best}}(n) = O(k * n)$$

$$T_{\text{worst}}(n) = O(k * n)$$



Exercise 3 – Task 3e: Remark

Comprehension question:

In the lecture slides (SL02/45), it is said, that Ω notation (non-tight, non-strict lower bound) is used for best case analysis. Shouldn't we therefore use Ω notation here for the best case?



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

Example Revisited: Asymptotic Complexity of C Code

In which asymptotic complexity class is the following C code function?

```
void functionA(int n)
{
    for (int i = n; i > 1; i -= 2)
    {
        printf(i);
    }
}
```

→ **$O(n)$**

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);
    }
}
```

→ $T(n) = T(n/2) + 1$

Running Time Analysis: Additional Example

Express the running time of the following C code function as a recurrence.

```
1 void bravo(int n, int a) {  
2     if (n < 1) {  
3         printf("%d", a);  
4         return;  
5     }  
6     bravo(n / 2, a - 1);  
7     bravo(n - 3, a + 1);  
8     bravo(n - 4, a + 2);  
9     bravo(n / 2, a - 2);  
10    printf("%d", a);  
11 }
```

Exercise 4, Tasks 1c and 1d

```
1 int alpha(int n) {  
2     if (n <= 2) {  
3         return 9;  
4     }  
5     else {  
6         return 5 + 7 * alpha(n / 3);  
7     }  
8     int t = 0;  
9     for (int i = 0; i < n; i++) {  
10         t = t + n;  
11     }  
12     printf("%d", t);  
13 }
```

$$T(n) = \begin{cases} T(\lfloor n/3 \rfloor) + \Theta(1), & \text{if } n > 2 \\ \Theta(1), & \text{if } n \leq 2 \end{cases}$$

$$T(n) \in \Theta(\log(n))$$

Note that lines 8 to 12 are dead code, i.e. they can never be executed.



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

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

Expanded recurrence

Term currently to be expanded

Repeated Substitution: Example

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

iteration	Expanded recurrence <i>Fill in the expanded term from right side as a substitution for the term that was expanded</i>	Term currently to be expanded <i>fill in the argument from left side everywhere you see n 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

$$n/2^x = 1$$

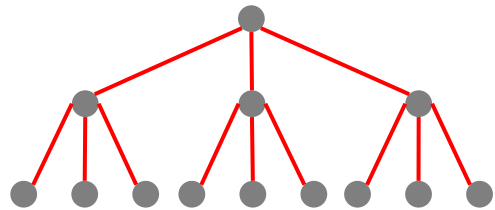
- From this: $n = 2^x$, $x = \log_2(n)$
- Now insert the x where we reach the base case into the **general expression** from above and then substitute the value given for $T(1)$:

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

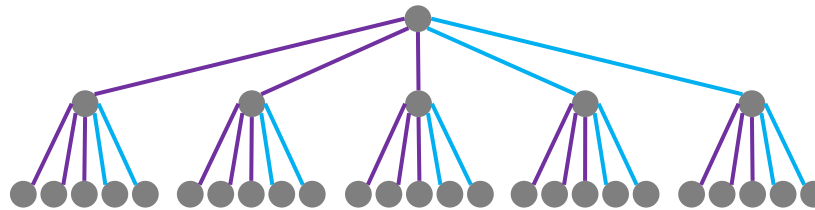
Recursion Tree Method

- 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](#).

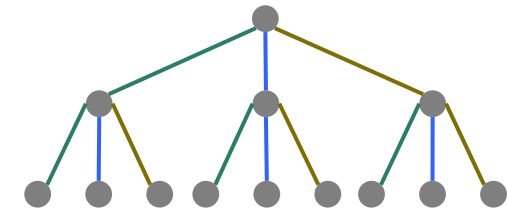
$$T(n) = 3 \cdot T(n/2)$$



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



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



- 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


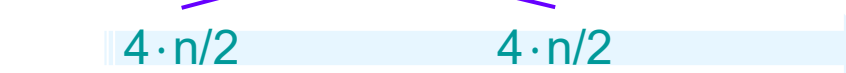
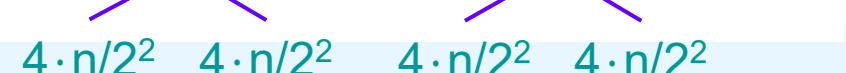

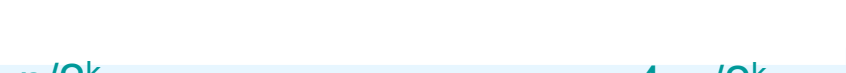

Consider the following recurrence relation: $T(n) = 2 \cdot T(n / 4) + n^2 \cdot \sqrt{n} \cdot \log(n)$

Draw the corresponding recursion tree.

Recursion Tree Method: Example

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

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

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		$\sum \dots$

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 4, Task 3a

$T(n) = 5T(n/5) + 7$, where $T(1) = 1$

$$\begin{aligned} T(n) &= 5T(n/5) + 7 \\ &= 5(5T(n/25) + 7) + 7 \\ &= 25T(n/25) + 5 \cdot 7 + 7 \\ &= 25T(n/25) + 5 \cdot 7 + 5^0 \cdot 7 \\ &= 25(5T(n/125) + 7) + 5 \cdot 7 + 5^0 \cdot 7 \\ &= 125T(n/125) + 25 \cdot 7 + 5 \cdot 7 + 5^0 \cdot 7 \\ &= 5^i T(n/5^i) + 5^{i-1} \cdot 7 + 5^{i-2} \cdot 7 + \dots + 5^1 \cdot 7 + 5^0 \cdot 7 \\ &= 5^i T(n/5^i) + 7 * (5^{i-1} + 5^{i-2} + \dots + 5^1 + 5^0) \\ &= \dots \end{aligned}$$

The base case will be reached when $T(n/5^{i_{max}}) = T(1)$, i.e. for $i_{max} = \log_5(n)$ (ignoring subtleties regarding rounding).

Exercise 4, Task 3a

In that case, according to the generalized form stemming from the repeated backwards substitution:

$$\begin{aligned} 5^{\log_5 n} T(n/5^{\log_5 n}) + 7 \cdot \sum_{i=0}^{i_{max}-1} 5^i &= 5^{\log_5 n} T(n/5^{\log_5 n}) + 7 \cdot \sum_{i=0}^{(\log_5 n)-1} 5^i \\ &= n \cdot T(1) + 7 \cdot \frac{1 - 5^{\log_5 n}}{1 - 5} \\ &= n + 7/4 \cdot n - 7/4 \end{aligned}$$

$$\Rightarrow T(n) \in \Theta(n)$$

Exercise 4, Task 3b

$$T(n) = 3T(n-2) + n, \text{ where } T(1) = 1$$

$$\begin{aligned} T(n) &= 3T(n-2) + n \\ &= 3(3T(n-2-2) + (n-2)) + n = 3^2T(n-2 \cdot 2) + 3 \cdot (n-2) + n = 9T(n-4) + 4n - 6 \\ &= 3(3(3T(n-2-2-2) + (n-4)) + (n-2)) + n \\ &= 3^3T(n-3 \cdot 2) + 3^2 \cdot (n-2 \cdot 2) + 3 \cdot (n-2) + n = 27T(n-6) + 13n - 42 \\ &= \dots \\ &= 3^kT(n-2k) + \sum_{i=1}^k 3^{i-1} \cdot (n-2 \cdot (i-1)) \\ &< 3^kT(n-2k) + n \cdot \sum_{i=1}^k 3^{i-1} \end{aligned}$$

The base case will be reached when $T(n-2k_{max}) = T(1)$, i.e. for $k_{max} = \lceil \frac{n-1}{2} \rceil$

As a simplification, only even values of n will be considered which allows to reformulate the above result as $k_{max} = n/2$ and $T(0) = T(1) = 1$ will be assumed.

This yields for the base case:

$$T(n) < 3^{\frac{n}{2}} \cdot T(1) + \Theta(n) \cdot \Theta(3^{\frac{n}{2}})$$

$$\Rightarrow \mathbf{T(n)} \in \mathbf{O(n\sqrt[2]{3^n})}$$

Exercise 4, Task 3c

$$\begin{aligned} T(n) &= T(\sqrt[2]{n}) + \log(n) = T(n^{\frac{1}{2}}) + \log(n) \\ &= T\left(\sqrt[2]{\sqrt[2]{n}}\right) + \log(\sqrt[2]{n}) + \log(n) = T(n^{\frac{1}{2^2}}) + \log(n^{\frac{1}{2}}) + \log(n) = T(n^{\frac{1}{4}}) + \frac{1}{2} \log(n) + \log(n) \\ &= T\left(\sqrt[2]{\sqrt[2]{\sqrt[2]{n}}}\right) + \log(\sqrt[2]{\sqrt[2]{n}}) + \log(\sqrt[2]{n}) + \log(n) = \\ &= T(n^{\frac{1}{2^3}}) + \log(n^{\frac{1}{2^2}}) + \log(n^{\frac{1}{2}}) + \log(n) = T(n^{\frac{1}{8}}) + \frac{1}{4} \log(n) + \frac{1}{2} \log(n) + \log(n) \\ &= \dots \\ &= T(n^{\frac{1}{2^k}}) + \sum_{i=0}^{k-1} \frac{1}{2^i} \log(n) = T(n^{\frac{1}{2^k}}) + \log(n) \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i \end{aligned}$$

Exercise 4, Task 3c

The base case will be reached when $T(n^{\frac{1}{2^{k_{max}}}}) = T(1)$, i.e. for $k_{max} = \log_2(n)$

The finite geometric series $\sum_{i=0}^k q^i$ can be bounded from above by the respective infinite geometric series $\sum_{i=0}^{\infty} q^i$. Since $|q| = \frac{1}{2} < 1$ this series converges to $\frac{1}{1-q} = 2$. (See task 4a for how a more strict upper boundary could be retrieved by using the formula for the finite geometric series.)

Therefore, for the base case:

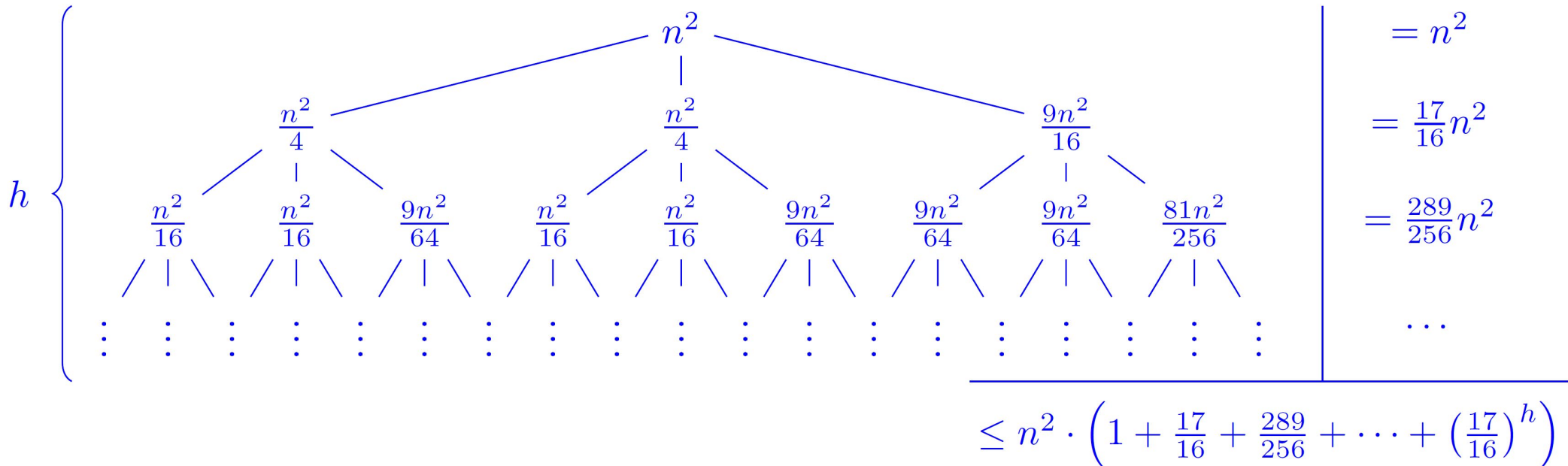
$$\begin{aligned} T(n) &\leq T(1) + \log(n) \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &\leq 1 + 2\log(n) \end{aligned}$$

$$\Rightarrow \log(n) < T(n) \leq 2 \cdot \log(n)$$

$$\Rightarrow T(n) \in O(\log(n))$$

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:

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

Reminder: 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.



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

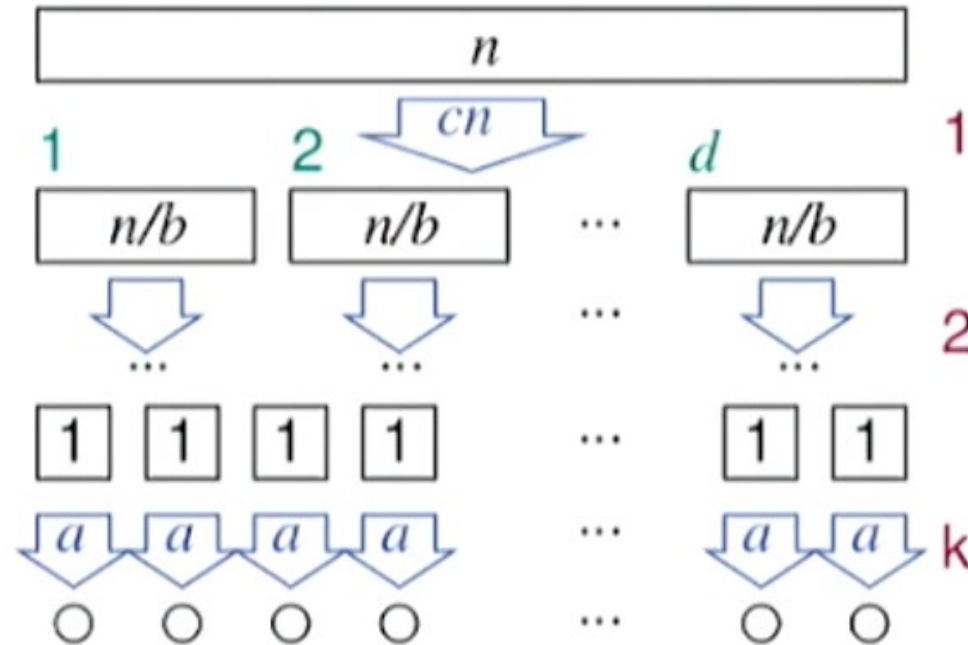
$$T(n) = a \cdot T(n/b) + f(n)$$

- 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 $f(n)$.»

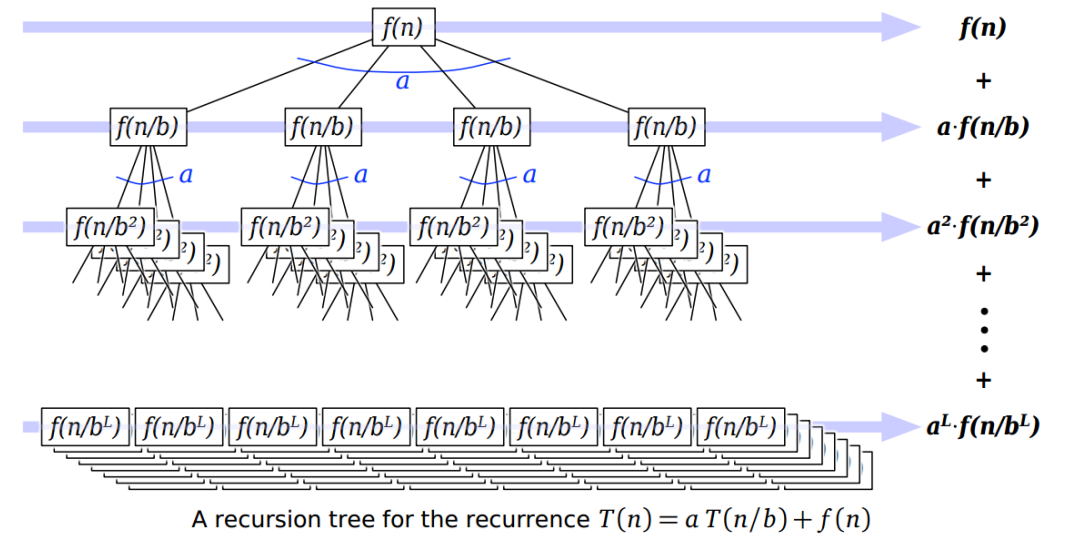
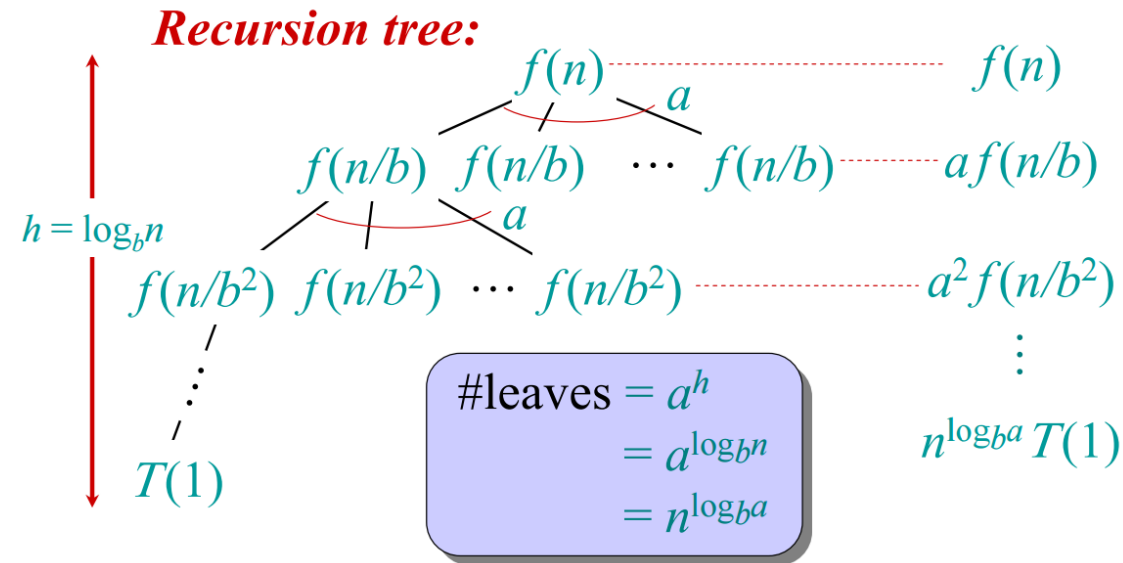
Master Method: Interpretation

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



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: $a < 1$
- $T_3(n) = 3 \cdot T_3(n/2^{-1}) + \log(n)$
→ 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

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 $T(\lceil n/b \rceil)$.

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, $T(n) \in \Theta(n^{2.5} \cdot \log(n))$

(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: $a = 36$, $b = 6$, $f(n) = 2 \cdot \sqrt{n} \cdot \log(n)$
case 1, $T(n) \in \Theta(n^2)$

(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: $a = 7$, $b = 11/3$, $f(n) = 21n + 8 \log(n!) + \sqrt{\log(n)} + n^2$
case 3, $T(n) \in \Theta(n^2)$

Substitution Method / Inductive Proof: Example

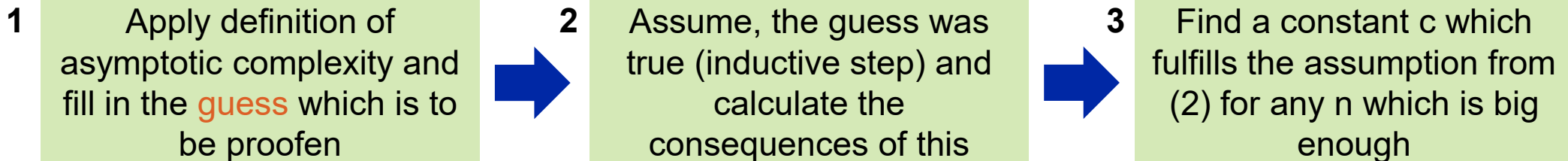
Task 1.4b from midterm 1 of FS 2016:

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:



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 4, Task 2a: Substitution Method

To show: $T(n) \in O(g(n))$ where $g(n) = n^2$

Hypothesis: $T(n) \leq c \cdot g(n) = c \cdot n^2$ for some constant $c > 0$

Inductive step: $T(n) = 3T(2n/3) + 5n^2 + 4n$
 $\leq 3 \cdot c \cdot g(2n/3) + 5n^2 + 4n$
 $= 3 \cdot c \cdot (2n/3)^2 + 5n^2 + 4n$
 $= (\frac{4c}{3} + 5)n^2 + 4n$

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

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

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

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

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

Inductive step: $T(n) = T(n/2) + 3T(3n/7) + 2n$
 $\leq c \cdot g(n/2) + 3 \cdot c \cdot g(3n/7) + 2n$
 $= c \cdot \log(n/2) + 3 \cdot c \cdot \log(3n/7) + 2n$
 $\not\leq c \cdot \log(n)$

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

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 $c > 0$

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

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 $T(n) \in O(g(n))$:

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

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



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
- Task 6: Sorting Coloured Circles
- Task 7: Modified Merge Sort Algorithm

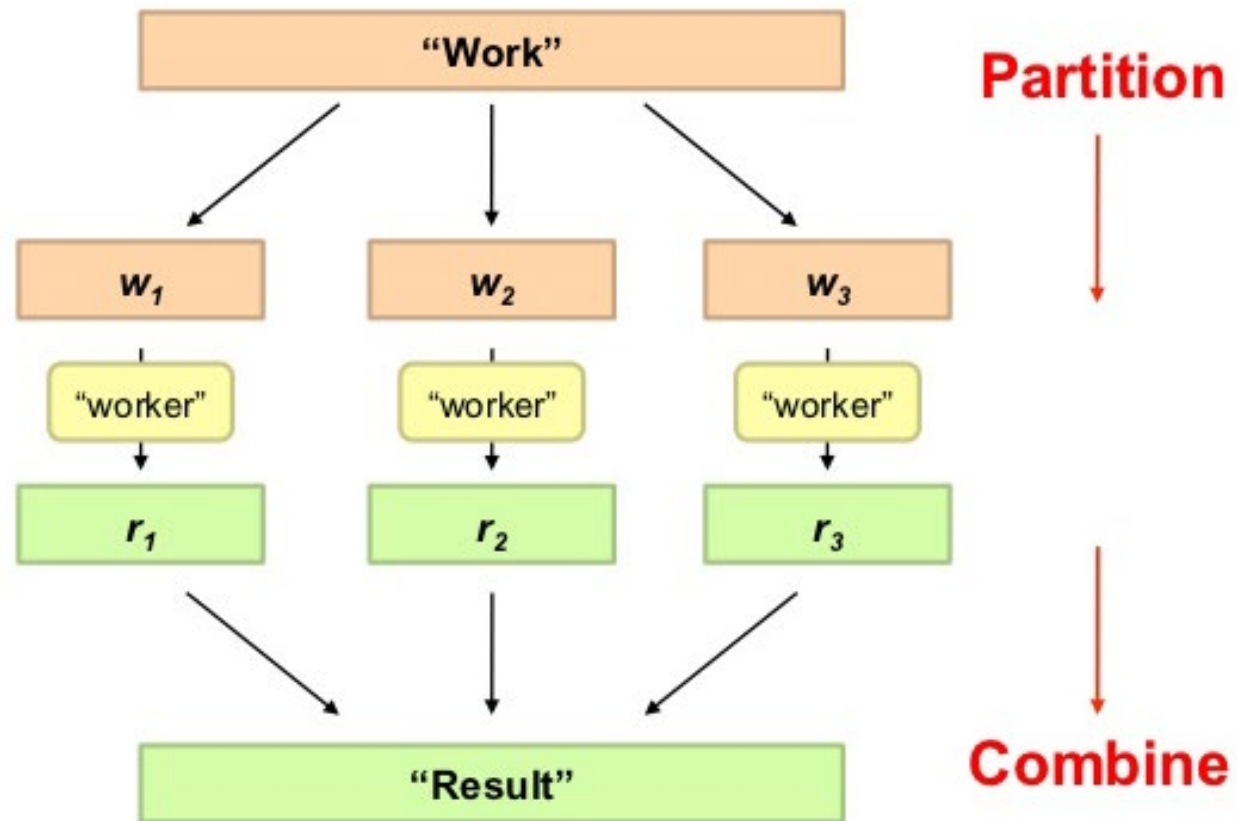


Divide and Conquer Algorithms

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



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?

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

Exercise 4, Task 6: Sorting Coloured Circles

 0,0	 0,1	 0,2	 0,3
 1,0	 1,1	 1,2	 1,3
 2,0	 2,1	 2,2	 2,3
 3,0	 3,1	 3,2	 3,3



 0,0	 0,1	 0,2	 0,3
 1,0	 1,1	 1,2	 1,3
 2,0	 2,1	 2,2	 2,3
 3,0	 3,1	 3,2	 3,3

Exercise 4, Task 6: Sorting Coloured Circles

Additional exercise: What is the recurrence which models the divide and conquer algorithm for solving this task?

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

Exercise 4, Task 7: 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.

(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.



Preview on Exercise 5

- Task 1: Heaps and Heapsort
- Task 2: Quicksort
- Task 3: Partitioning in Quicksort
- Task 4: Tasks from Previous Midterm



**Universität
Zürich** ^{UZH}

Institut für Informatik

Wrap-Up

- Summary
- Feedback
- Outlook
- Questions



Wrap-Up

- Summary



Outlook on Next Thursday's Lab Session

Next tutorial: Wednesday, 30.03.2022, 14.00 h, BIN 0.B.06

Topics:

- Review of remaining parts of Exercise 4
- Review of Exercise 5
- Heaps, Heapsort
- Quicksort
- Preview on Exercise 6
- ...
- ... (your wishes)



**Universität
Zürich** ^{UZH}

Institut für Informatik

Questions?



Thank you for your attention.