



Tutorial Session 4

Tuesday, 13th of March 2020

Discussion of Exercise 3, Preview on Exercise 4

Running Time Analysis, Invariants, Recurrences

14.15 – 15.45

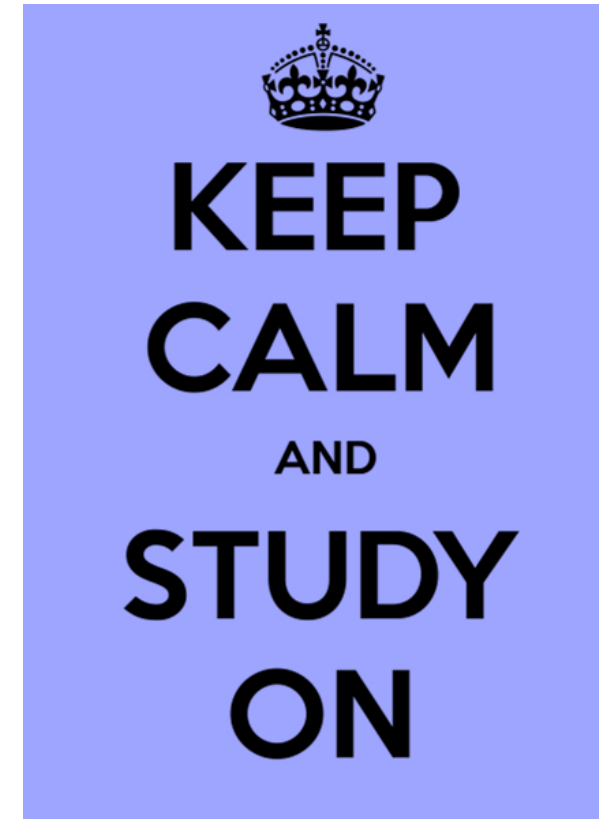
Y35-F-32



Agenda

- Administrative Affairs
- Discussion of Exercise 3
- *poss.* Loop Invariants
- *poss.* Recurrences
- Preview on Exercise 4

The Situation





The Situation

- As of today, our lab sessions will be recorded and provided as podcast on OLAT.
- I will try to adapt the exercise sessions such that they are suitable for recording. Although I cannot give any guarantee that they will work technically and/or that each and everything will be completely comprehensible (e.g. writing on the blackboard).
- As you all are aware, the situation is quite fluid at the moment. That's as true for the teaching staff as it is true for you.
- University has announced further information regarding the situation for today at 15.00.



Review of Exercise 3

- Task 1: Running Time Analysis et al.
- Task 2: Asymptotic Complexities
- Task 3: Special Case Analysis

Exercise 3 – Task 1a: Implementing Pseudocode in C

Algorithm `whatDoesItDo(A,k)` gets an array `A[1..n]` of n integers and an integer number k as an input.

Algo: `WHATDOESITDO(A,k)`

```
1 sum = 0;
2 for i = 1 to k do
3     mini = i;
4     for j = i + 1 to length(A) do
5         if A[j] < A[mini] then
6             mini = j;
7     sum = sum + A[mini];
8     swp = A[i];
9     A[i] = A[mini];
10    A[mini] = swp;
11 return sum
```

a) Implement the algorithm as a C program that reads the elements of `A` and prints the result.

Exercise 3 – Task 1a: Implementing Pseudocode in C

```

1  int sumKLowest(int a[], int k, int n) {
2      int i, j, swp;
3      int mini;
4      int sum = 0;
5      for(i = 0; i < k; i++) {
6          mini = i;
7          for(j = i + 1; j < n; j++) {
8              if(a[j] < a[mini]) mini = j;
9          }
10         sum += a[mini];
11         swp = a[i];
12         a[i] = a[mini];
13         a[mini] = swp;
14     }
15     return sum;
16 }
```

Algo: WHATDOESITDo(A,k)

```

1  sum = 0;
2  for i = 1 to k do
3      mini = i;
4      for j = i + 1 to length(A) do
5          if A[j] < A[mini] then
6              mini = j;
7      sum = sum + A[mini];
8      swp = A[i];
9      A[i] = A[mini];
10     A[mini] = swp;
11 return sum
```

- Indexing starts with 1 in pseudocode and with 0 in C code.
- In C, an additional parameter is needed for the function to know the size of the array.

Exercise 3 – Task 1b: What Does It Do?

$A =$

42	3	8	2	7	11	7
1	2	3	4	5	6	7

This algorithm calculates the sum of the k -lowest integers in array $A[1..n]$. For example, given the input array is $A = [12, 4, 10, 2, 8]$, if $k = 3$, the result is $2 + 4 + 8 = 14$ whereas, if $k = 4$, the result is $2 + 4 + 8 + 10 = 24$

Algo: WHATDOESITDo(A, k)

```
1 sum = 0;
2 for  $i = 1$  to  $k$  do
3   mini = i;
4   for  $j = i + 1$  to  $length(A)$  do
5     if  $A[j] < A[mini]$  then
6       mini = j;
7   sum = sum +  $A[mini]$ ;
8   swp =  $A[i]$ ;
9    $A[i] = A[mini]$ ;
10   $A[mini] = swp$ ;
11 return sum
```

Exercise 3 – Task 1c: Exact Running Time Analysis

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>mini</i> := <i>i</i>	k	c_3
for <i>j</i> := <i>i</i> + 1 to <i>n</i> do	$\left(kn - \frac{k(k+1)}{2}\right)^* + 2k^{**}$	c_4
if $A[j] < A[mini]$ then	$kn - \frac{k(k+1)}{2}$	c_5
<i>mini</i> := <i>j</i>	$\alpha \left(kn - \frac{k(k+1)}{2}\right)^{***}$	c_6
<i>sum</i> := <i>sum</i> + $A[mini]$	k	c_7
<i>swp</i> := $A[i]$	k	c_8
$A[i]$:= $A[mini]$	k	c_9
$A[mini]$:= <i>swp</i>	k	c_{10}
return <i>sum</i>	1	c_{11}

$$* (n - 2 + 1) + (n - 3 + 1) + \dots + (n - k) = \sum_{q=1}^k (n - q) = kn - \frac{k(k+1)}{2}$$

** k times for $i + 1$ and k times for termination condition

*** $0 \leq \alpha \leq 1$

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

Exercise 3 – Task 1d and 1e: Best and Worst Case, Influence of k

d) Determine the best and the worst case of the algorithm. What is the running time and asymptotic complexity in each case?

Best case

$$\alpha = 0, k = 1,$$

$$T_{\text{best}}(n) = c_1 + 2c_2 + c_3 + c_4(n + 1) + c_5(n - 1) + 0 + c_7 + c_8 + c_9 + c_{10} + c_{11}$$

Worst case

$$\alpha = 1, k = n,$$

$$T_{\text{worst}}(n) = c_1 + c_2(n + 1) + c_3n + c_4\left(\frac{n^2}{2} + \frac{3}{2}n\right) + c_5\left(\frac{n^2}{2} - \frac{n}{2}\right) + c_6\left(\frac{n^2}{2} - \frac{n}{2}\right) + (c_7 + c_8 + c_9 + c_{10})n + c_{11}$$

Asymptotic complexity of best and worst case $T_{\text{best}}(n) = \Theta(n)$

$$T_{\text{worst}}(n) = \Theta(n^2)$$

e) What influence has the parameter k in the asymptotic complexity?

The parameter k has no direct influence on asymptotic complexity because it is fixed for the two cases and defines best and worst case.

Exercise 3 – Task 2: Asymptotic Complexity

Calculate the asymptotic tight bound for the following functions and rank them by their order of growth (lowest first). Clearly work out the calculation steps in your solution.

- $f_1(n) = n^n + 2^{2n} + 13^{124}$ $\in \Theta(n^n)$
- $f_2(n) = \log(14 \cdot (n - 1) \cdot n^{3n+2})$ $\in \Theta(n \cdot \log(n))$
- $f_3(n) = 4^{\log_2(n)}$ $\in \Theta(n^2)$
- $f_4(n) = 12\sqrt{n} + 10^{223} + \log(5^n)$ $\in \Theta(n)$
- $f_5(n) = n^2 \cdot \log(n + 1) + n \cdot \log(n^2) + 0.5n$ $\in \Theta(n^2 \cdot \log(n))$
- $f_6(n) = 7n^4 + 100n \cdot \log(n) + \sqrt{32} + n$ $\in \Theta(n^4)$
- $f_7(n) = \log(\min(n, \sqrt{n}))$ $\in \Theta(\log(n))$
- $f_8(n) = \log^2(n) + 50\sqrt{n} + \log(n)$ $\in \Theta(n^{0.5})$
- $f_9(n) = (n + 3)!$ $\in \Theta((n + 3)!)$
- $f_{10}(n) = 2 \cdot \log(6^{\log(n^2)}) + \log(\pi n^2) + n^3$ $\in \Theta(n^3)$

Ranking: (slowest growing) $f_7, f_8, f_4, f_2, f_3, f_5, f_{10}, f_6, f_9, f_1$ (fastest growing)



Asymptotic Complexity: Additional Practice

Additional examples for practice can be found here:

https://www.physik.uzh.ch/~vogelch/algodat/practice/asymptotic_complexity.html

Exercise 3 – Task 3a: Special Case Analysis

Given two strings A and B, develop an algorithm that checks if B is a substring of A and, if so, returns the number of occurrences of B in A.

a) Specify all the special cases that need to be considered and provide examples of the input data for each of them.

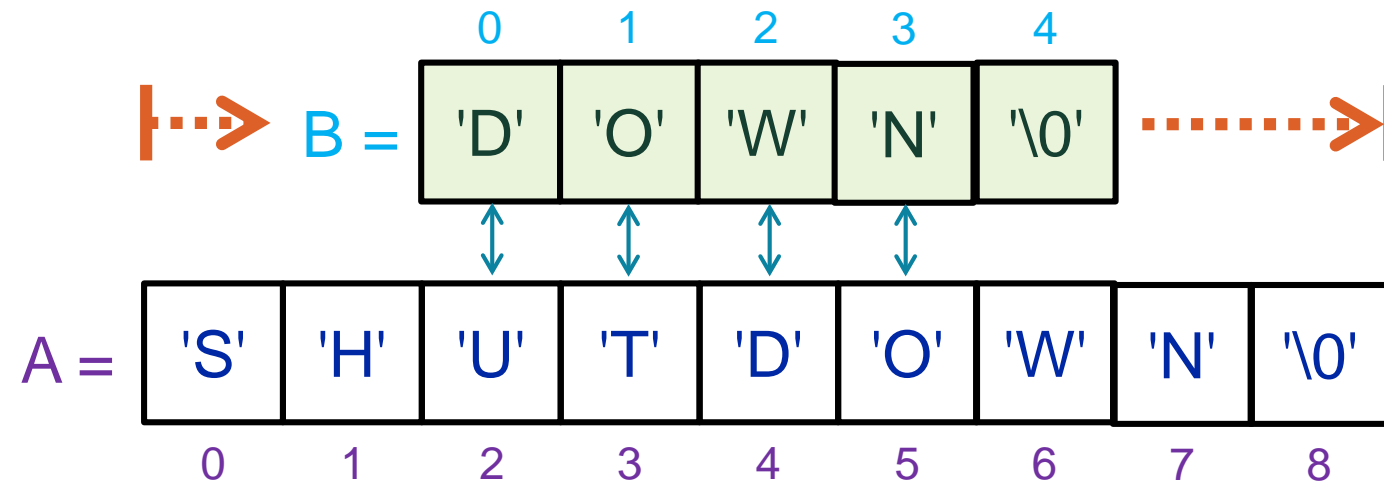
“Standard case”: A is longer than B and A contains B at most once.

#	Case	A	B	result	
1	A is empty	␣	mis	0	} at least one string is non-existent (null) or empty
2	B is empty	understanding	␣	0	
3	A is NULL	NULL	mis	0	
4	B is NULL	understanding	NULL	0	
5	B is longer than A	under	understanding	0	← length(B) > length(A)
6	A = B	understanding	understanding	1	← A == B
7	A contains several B	abbababba	abba	2	← result >= 2, non overlapping
8	B overlapping in A	aaaa	aa	3	← result >= 2, overlapping

Note that there is a difference between «empty» and «NULL» (similar to the situation in Python where a string can be empty or «None»).

Exercise 3 – Task 3b: Special Case Analysis

Idea of algorithm: Take potential substring B and «slide» it along the target string A. For each position of the target string, compare the respective character to the character at each position of the substring (abort as soon as a non-matching character is discovered).



Exercise 3 – Task 3b: Special Case Analysis

```
int substrings(char A[], char B[]) {  
    if(A == NULL || B == NULL) {  
        return 0;  
    }  
    if(A[0] == '\\0' || B[0] == '\\0') {  
        return 0;  
    }  
  
    int sizeB = 0;  
    while(B[sizeB] != '\\0') {  
        sizeB++;  
    }  
  
    int numOccurrences = 0;  
    int currentPosition = 0;  
    int matchingChars = 0;  
}
```

early abort { at least one string is non-existent (null) or empty

“setup” { count number of characters in string B[]

{ initialize book-keeping variables

Exercise 3 – Task 3b

slide along
target string

```
while(A[currentPosition] != '\0') {
```

```
    matchingChars = 0;
```

```
    while (B[matchingChars] != '\0' &&
```

```
        A[currentPosition + matchingChars] == B[matchingChars]) {
```

```
        matchingChars++;
```

```
    }
```

```
    if (matchingChars == sizeB) {
```

```
        printf("(%d,%d) ", currentPosition + 1, currentPosition + sizeB);
```

```
        numOccurrences++;
```

```
    }
```

```
    currentPosition++;
```

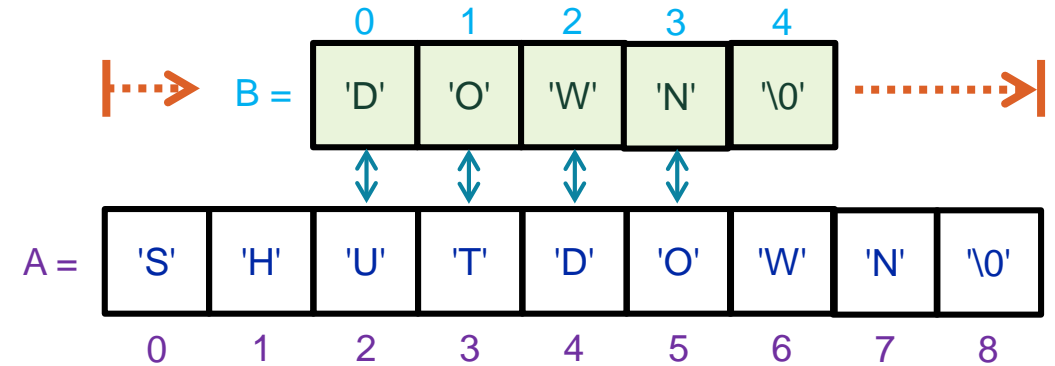
```
}
```

```
return numOccurrences;
```

```
}
```

compare
characters

hit found if number of
character matches
equals length(B)





Algorithm Running Time Analysis

- General Principles
- Examples Solved



Methods for Determining the Number of Executions of for Loops in Asymptotic Running Time Analysis

- Multiplicative combination of algorithm parts (and using «end – start + 1» et al.)
- «Table method»: write down the evolution of the loop variable in a table to gain an understanding about the number of executions (probably not possible for more than two levels of nesting)
- Write loop as a nested multiple sum and solve it (requires a bit of knowledge how to manipulate sums).

Algorithm Running Time Analysis: Example 1

Task 1 of assignment 2 from spring semester 2016:

Given an unsorted array $A[1..n]$ of integers and an integer k , the adjacent algorithm calculates the maximum value of every contiguous subarray of size k .

Perform an exact analysis of the running time of the following algorithm.

Also determine the best and the worst case of the algorithm and what the running time and asymptotic complexity is in each case.

Algorithm: $\text{findKmax}(A, k, n)$

```
for  $i = 1$  to  $n - k + 1$  do
     $\text{max} = A[i]$ 
    for  $j = 1$  to  $k - 1$  do
        if  $A[i + j] > \text{max}$  then
             $\text{max} = A[i + j]$ 
    print( $\text{max}$ )
```

Algorithm Running Time Analysis: Example 1 Solution

Algorithm: findKmax(A, k, n)	Cost	Number of times executed
1 for i = 1 to n - k + 1 do	C_1	
2 max = A[i]	C_2	
3 for j = 1 to k - 1 do	C_3	
4 if A[i + j] > max then	C_4	
5 max = A[i + j]	C_5	
6 print(max)	C_6	

Algorithm Running Time Analysis: Example 1 Solution

Algorithm: findKmax(A, k, n)	Cost	Number of times executed
1 for i = 1 to n - k + 1 do	c_1	$n - k + 2$
2 max = A[i]	c_2	$n - k + 1$
3 for j = 1 to k - 1 do	c_3	$k \cdot (n - k + 1)$
4 if A[i + j] > max then	c_4	$(k - 1) \cdot (n - k + 1)$
5 max = A[i + j]	c_5	$\alpha \cdot (k - 1) \cdot (n - k + 1); \quad 0 \leq \alpha \leq 1$
6 print(max)	c_6	$n - k + 1$

$$\begin{aligned}
 T(n) &= c_1 \cdot (n - k + 2) + c_2 \cdot (n - k + 1) + c_3 \cdot k \cdot (n - k + 1) + c_4 \cdot (k - 1) \cdot (n - k + 1) + \\
 &\quad + c_5 \cdot \alpha \cdot (k - 1) \cdot (n - k + 1) + c_6 \cdot (n - k + 1) = \\
 &= c_1 + (c_1 + c_2 - c_4 + c_6 - c_5 \cdot \alpha + (c_3 + c_4 + c_5 \cdot \alpha) \cdot k) \cdot (n - k + 1) \quad \rightarrow \text{in worst case } O(n^2)
 \end{aligned}$$

Algorithm Running Time Analysis: Example 2

Exercise 2.3 from midterm 1 of spring semester 2016:

Perform an exact analysis of the running time of the following algorithm and determine its asymptotic complexity.

Algorithm: $\text{alg}(A, n)$

```
1  for  $i = 1$  to  $\lfloor n / 2 \rfloor$  do
2       $\text{min} = i$ 
3       $\text{max} = n - i + 1$ 
4      if  $A[\text{min}] > A[\text{max}]$  then
5           $\text{exchange } A[\text{min}] \text{ and } A[\text{max}]$ 
6      for  $j = i + 1$  to  $n - i$  do
7          if  $A[j] < A[\text{min}]$  then
8               $\text{min} = j$ 
9          if  $A[j] > A[\text{max}]$  then
10              $\text{max} = j$ 
11      $\text{exchange } A[i] \text{ and } A[\text{min}]$ 
12      $\text{exchange } A[n - i + 1] \text{ and } A[\text{max}]$ 
```

Algorithm Running Time Analysis: Example 2 Solution

Algorithm: $\text{alg}(A, n)$		Cost	Number of times executed
1	for $i = 1$ to $\lfloor n / 2 \rfloor$ do	C_1	
2	$\text{min} = i$	C_2	
3	$\text{max} = n - i + 1$	C_3	
4	if $A[\text{min}] > A[\text{max}]$ then	C_4	
5	exchange $A[\text{min}]$ and $A[\text{max}]$	C_5	
6	for $j = i + 1$ to $n - i$ do	C_6	
7	if $A[j] < A[\text{min}]$ then	C_7	
8	$\text{min} = j$	C_8	
9	if $A[j] > A[\text{max}]$ then	C_9	
10	$\text{max} = j$	C_{10}	
11	exchange $A[i]$ and $A[\text{min}]$	C_{11}	
12	exchange $A[n - i + 1]$ and $A[\text{max}]$	C_{12}	

Algorithm Running Time Analysis: «Table Method» Example

The following example shows how to analyse the number executions of the body of the inner loop (lines 7 to 10) of the algorithm $\text{alg}(A, n)$ using a table to track the evolution of the number of executions depending on the value of the outer loop variable.

Outer loop variable value $i =$	Inner loop variable range $j = i+1 \dots n-i$	Number of executions of the inner loop body

Algorithm: $\text{alg}(A, n)$

```

1 for  $i = 1$  to  $\lfloor n/2 \rfloor$  do
2    $min = i$ ;
3    $max = n - i + 1$ ;
4   if  $A[min] > A[max]$  then
5     exchange  $A[min]$  and  $A[max]$ ;
6   for  $j = i + 1$  to  $n - i$  do
7     if  $A[j] < A[min]$  then
8        $min = j$ ;
9     if  $A[j] > A[max]$  then
10       $max = j$ ;
11  exchange  $A[i]$  and  $A[min]$ ;
12  exchange  $A[n - i + 1]$  and  $A[max]$ ;

```


Algorithm Running Time Analysis: «Table Method» Example

The following example shows how to analyse the number executions of the body of the inner loop (lines 7 to 10) of the algorithm $\text{alg}(A, n)$ using a table to track the evolution of the number of executions depending on the value of the outer loop variable.

Outer loop variable value $i =$	Inner loop variable range $j = i+1 \dots n-i$	Number of executions of the inner loop body
1	2 ... $n-1$	$n - 2$
2	3 ... $n-2$	$n - 4$
3	4 ... $n-3$	$n - 6$
\vdots	\vdots	\vdots
k	$k+1 \dots n-k$	$n - 2 \cdot k$
\vdots	\vdots	\vdots
$\lfloor n/2 \rfloor$	$\lfloor n/2 \rfloor + 1 \dots \lfloor n/2 \rfloor$	0

Algorithm: $\text{alg}(A, n)$

```

1 for  $i = 1$  to  $\lfloor n/2 \rfloor$  do
2    $min = i$ ;
3    $max = n - i + 1$ ;
4   if  $A[min] > A[max]$  then
5     exchange  $A[min]$  and  $A[max]$ ;
6   for  $j = i + 1$  to  $n - i$  do
7     if  $A[j] < A[min]$  then
8        $min = j$ ;
9     if  $A[j] > A[max]$  then
10       $max = j$ ;
11  exchange  $A[i]$  and  $A[min]$ ;
12  exchange  $A[n - i + 1]$  and  $A[max]$ ;

```

Algorithm Running Time Analysis: «Table Method» Example

The total number of execution within both of the nested loop (body) is the sum of the executions of the inner loop over all iterations of the outer loop:

$$\sum_{k=1}^{\lfloor n/2 \rfloor} n - 2 \cdot k$$

This sum can be resolved as follows:

$$\sum_{k=1}^{\lfloor n/2 \rfloor} n - 2 \cdot k = \sum_{k=1}^{\lfloor n/2 \rfloor} n - \sum_{k=1}^{\lfloor n/2 \rfloor} 2 \cdot k = n \cdot \sum_{k=1}^{\lfloor n/2 \rfloor} 1 - 2 \cdot \sum_{k=1}^{\lfloor n/2 \rfloor} k = n \cdot \lfloor n/2 \rfloor - 2 \cdot \frac{\lfloor n/2 \rfloor \cdot (\lfloor n/2 \rfloor + 1)}{2}$$

Algorithm Running Time Analysis: «Nested Multiple Sum» Example

$$\sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{j=i+1}^{n-i} 1$$

Algorithm: $\text{alg}(A, n)$

```
1 for  $i = 1$  to  $\lfloor n/2 \rfloor$  do
2    $min = i$ ;
3    $max = n - i + 1$ ;
4   if  $A[min] > A[max]$  then
5     exchange  $A[min]$  and  $A[max]$ ;
6   for  $j = i + 1$  to  $n - i$  do
7     if  $A[j] < A[min]$  then
8        $min = j$ ;
9     if  $A[j] > A[max]$  then
10       $max = j$ ;
11  exchange  $A[i]$  and  $A[min]$ ;
12  exchange  $A[n - i + 1]$  and  $A[max]$ ;
```



Algorithm Running Time Analysis: Summation Rules

- General Rules and «Non-Rules»
- Arithmetic Series
- Series of Polynomial Expressions
- Geometric Series
- Harmonic Series
- Arithmetico-Geometric Series
- Decomposing Sums
- Changing Summation Indexes
- Double Sums / Multiple Sums / Nested Sums

Summations: General Rules / Simple Manipulations

- Summation of constants are equivalent to multiplication (take care about the number of summands):

$$\sum_{k=1}^n c = c + c + \dots + c = n \cdot c \quad (\text{for any constant } c)$$

- Constant multiplication factors in summations can be «pulled out» of the summations:

$$\sum_{k=1}^n c \cdot a_k = c \cdot a_1 + c \cdot a_2 + \dots + c \cdot a_n = c \cdot \sum_{k=1}^n a_k \quad (\text{for any constant } c)$$

- Sums in summations can be split up into separate summations (with both the same indexing):

$$\sum_{k=1}^n a_k + b_k = (a_1 + b_1) + (a_2 + b_2) + \dots + (a_n + b_n) = \sum_{k=1}^n a_k + \sum_{k=1}^n b_k$$

Summations: «Non-Rules»

- Multiplication:

$$\sum_{k=1}^n (a_k \cdot b_k) \neq \left(\sum_{k=1}^n a_k \right) \cdot \left(\sum_{k=1}^n b_k \right)$$

- Division:

$$\sum_{k=1}^n (a_k / b_k) \neq \frac{\left(\sum_{k=1}^n a_k \right)}{\left(\sum_{k=1}^n b_k \right)}$$



Summations: Arithmetic Series

A summation of the form

$$\sum_{k=0}^n a_0 + k \cdot d$$

is called an (finite) arithmetic series with $d = a_{k+1} - a_k = \text{const.}$

The finite arithmetic series sums up to $(n + 1) \cdot \frac{a_0 + a_n}{2}$ where $a_n = a_0 + n \cdot d$



Summations: Arithmetic Series

There is one particularly famous and important (and often used) case of the arithmetic series:

$$\sum_{k=0}^n k = \sum_{k=1}^n k = \frac{n \cdot (n + 1)}{2}$$

(Gauß'sche Summenformel, $a_0 = 0$, $d = 1$)

Summations: Polynomial Expressions

- Sum of squares:

$$\sum_{k=1}^n k^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} \in \Theta(n^3)$$

- Sum of cubes:

$$\sum_{k=1}^n k^3 = 1^3 + 2^3 + 3^3 + \dots + n^3 = \left(\sum_{k=1}^n k \right)^2 = \left(\frac{n \cdot (n+1)}{2} \right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4} \in \Theta(n^4)$$

- Sum of fourths:

$$\sum_{k=1}^n k^4 = 1^4 + 2^4 + 3^4 + \dots + n^4 = \frac{n \cdot (n+1) \cdot (2n+1) \cdot (3n^2 + 3n - 1)}{30} = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} + \frac{n}{30} \in \Theta(n^5)$$



Summations: Polynomial Expressions

- Sum of the square of a sum:

$$\sum_{k=1}^n (a_k + b_k)^2 = (a_1 + b_1)^2 + (a_2 + b_2)^2 + \dots + (a_n + b_n)^2 = \sum_{k=1}^n a_k^2 + 2 \cdot \sum_{k=1}^n a_k \cdot b_k + \sum_{k=1}^n b_k^2$$

Summations: 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 $1 / (1 - a)$ iff $|a| < 1$, otherwise it diverges.

Summations: Harmonic Series

A summation of the form

$$\sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$$

is called an (finite) harmonic series.

There is no closed formula for these sums.

Note that, although the associated sequence is a zero sequence (Nullfolge; thus its terms come always closer to zero), the *infinite* harmonic series is *divergent*.

Summations: 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:

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

Example: C Code Running Time Analysis with Series

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

Write the number of executions of line 6 as a summation formula.

What will be the return value of functionK(100)?

```
int functionK(int n) {  
    int z = 0;  
    for (int i = 1; i <= n; i++) {  
        for (int j = 1; j <= n; j++) {  
            for (int k = 1; k <= j; k++) {  
                z = z + 1;  
            }  
        }  
    }  
    return z;  
}
```

→ $O(n^3)$

$$\begin{aligned} \sum_{i=1}^n \sum_{j=0}^n \sum_{k=1}^j 1 &= \\ &= \sum_{i=1}^n \sum_{j=0}^n j = \sum_{i=1}^n \frac{n \cdot (n+1)}{2} = \\ &= n \cdot \frac{n \cdot (n+1)}{2} = \frac{1}{2} \cdot (n^3 + n^2) = f_K(n) \end{aligned}$$

→ $f_K(100) = 505000$

Summations: Decomposing Sums

Indexes of sums can be rewritten / sums can be «taken apart» as in the following example:

$$\sum_{k=x}^{n-x} k = \sum_{k=1}^{n-x} k - \sum_{k=1}^{x-1} k$$

↑
sum up «too much», i.e.
start summing up from 1
instead of starting from x

↑
remove what has been
summed up too much, i.e.
numbers from 1 to x - 1



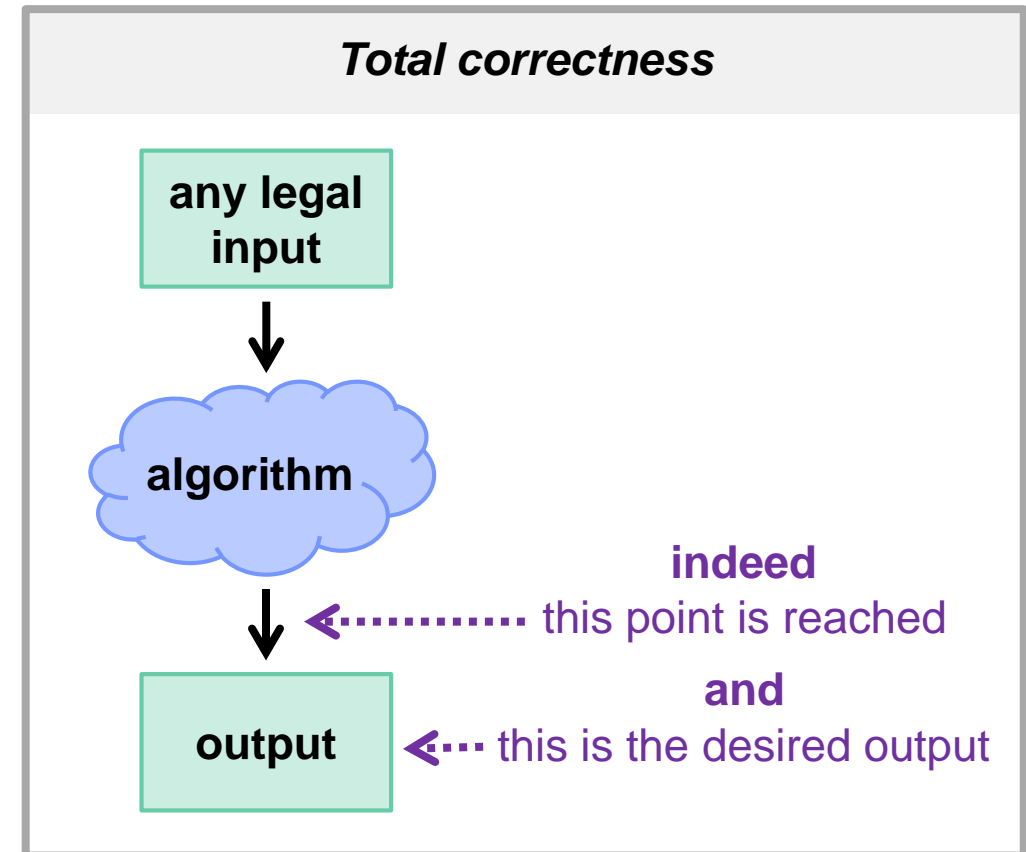
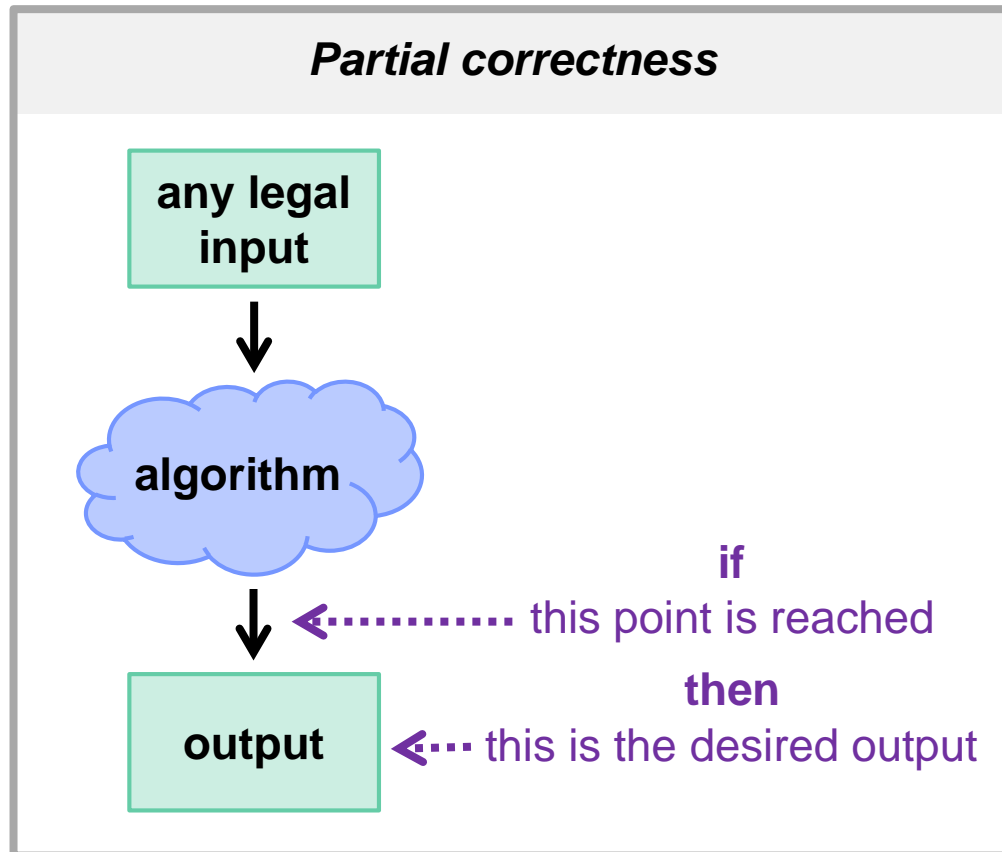
**Universität
Zürich** UZH

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.

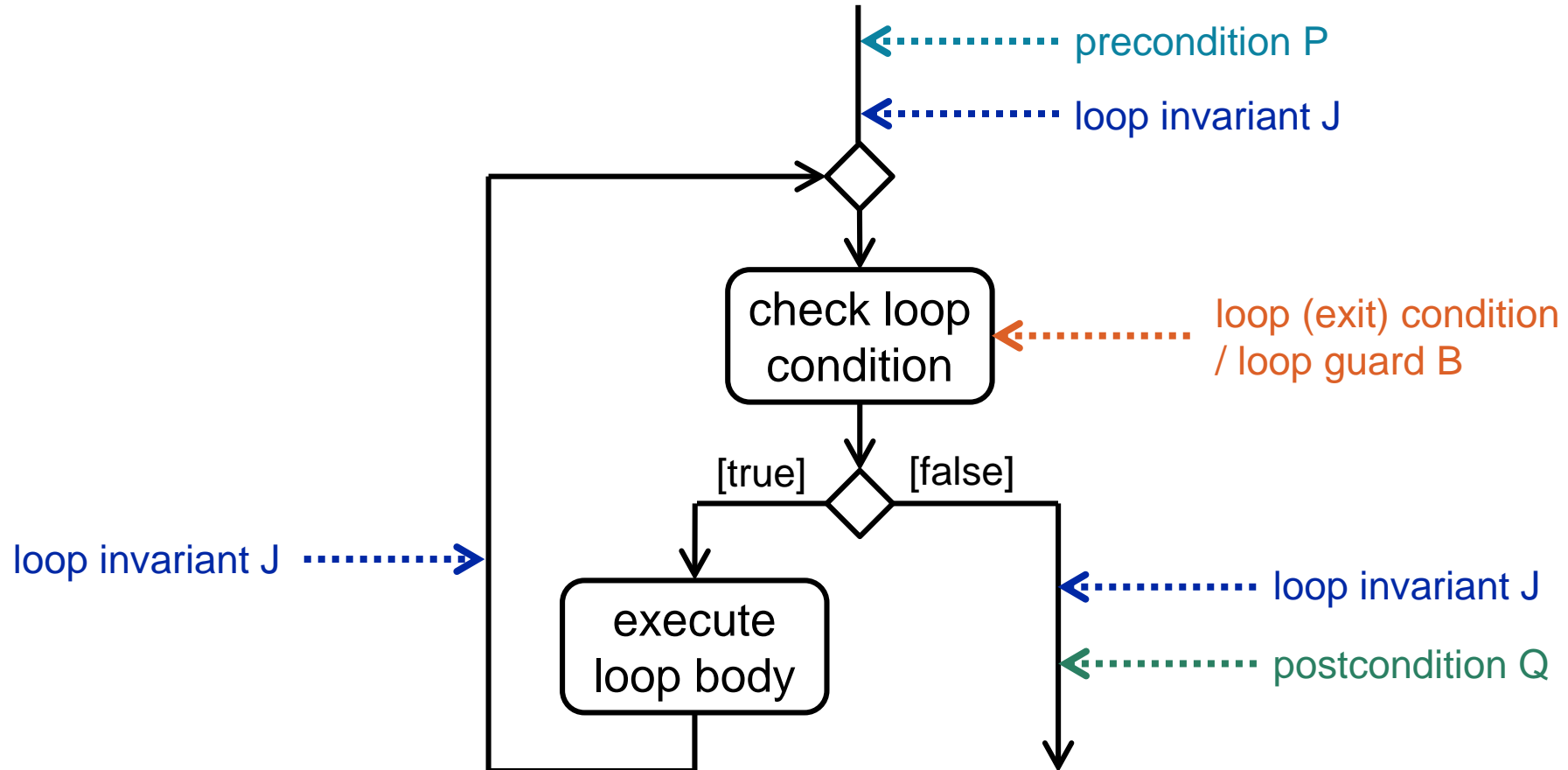
If the loop invariant was found and formulated using natural language, it has to be written using formal language (predicate logic). This part is mainly a matter of training.

Precondition and Postcondition

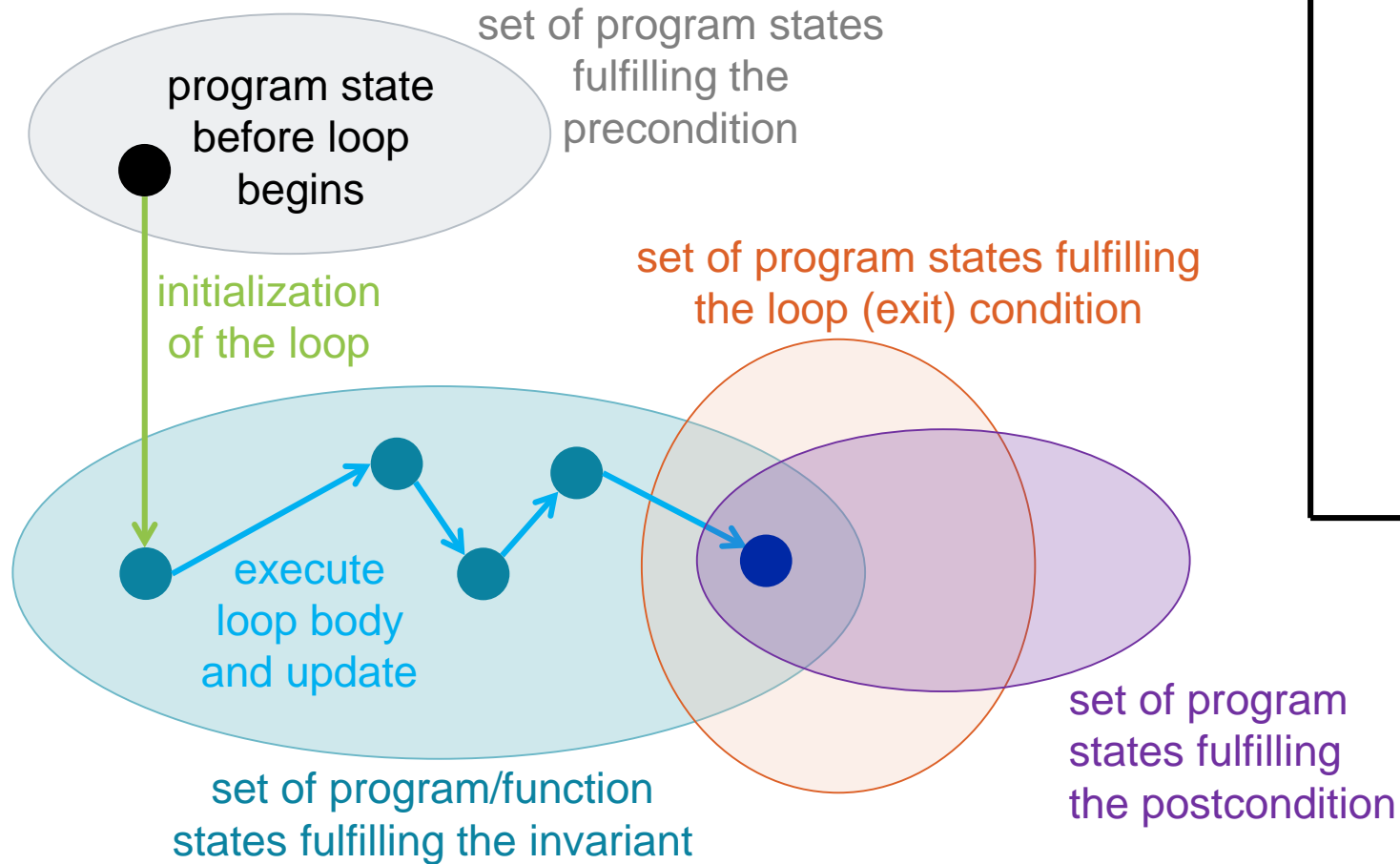
Precondition and postcondition are logical expressions which can be evaluated to true or false.

- 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

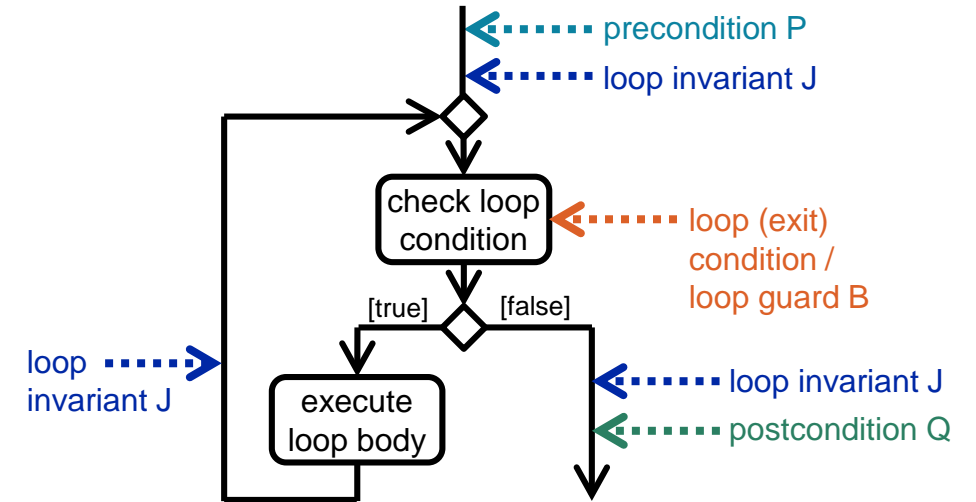


Example:

```
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 (π) 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.



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 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

$$n/2^x = 1$$

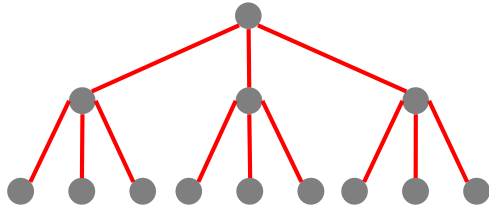
- From this: $n = 2^x$, $x = \log_2(n)$
- Now insert the x where we reach the base case into the **general expression** from above and then substitute the value given for $T(1)$:

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

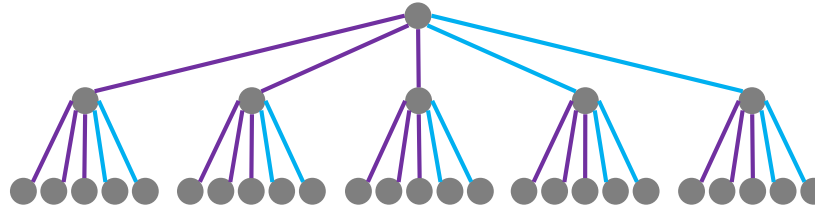
Recursion Tree Method

- 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](#).

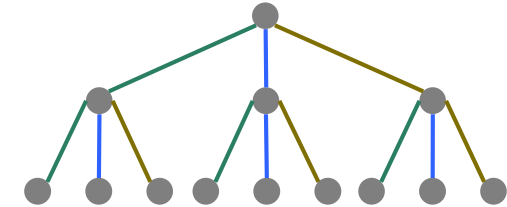
$$T(n) = 3 \cdot T(n/2)$$



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



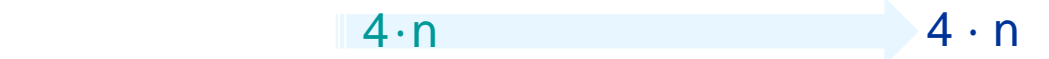



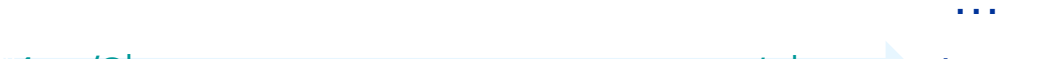


$$T(n) = 1 \cdot T(n-1) + 1 \cdot T(n-2) + 1 \cdot T(n-3)$$



- 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.

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:

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

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



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

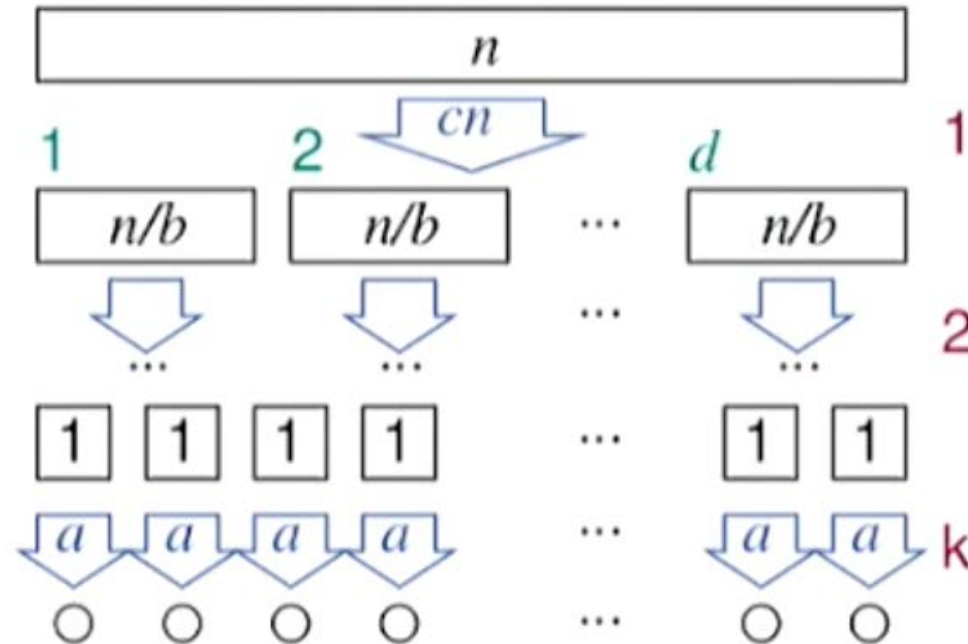
$$T(n) = a \cdot T(n/b) + f(n)$$

- 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 $f(n)$.»

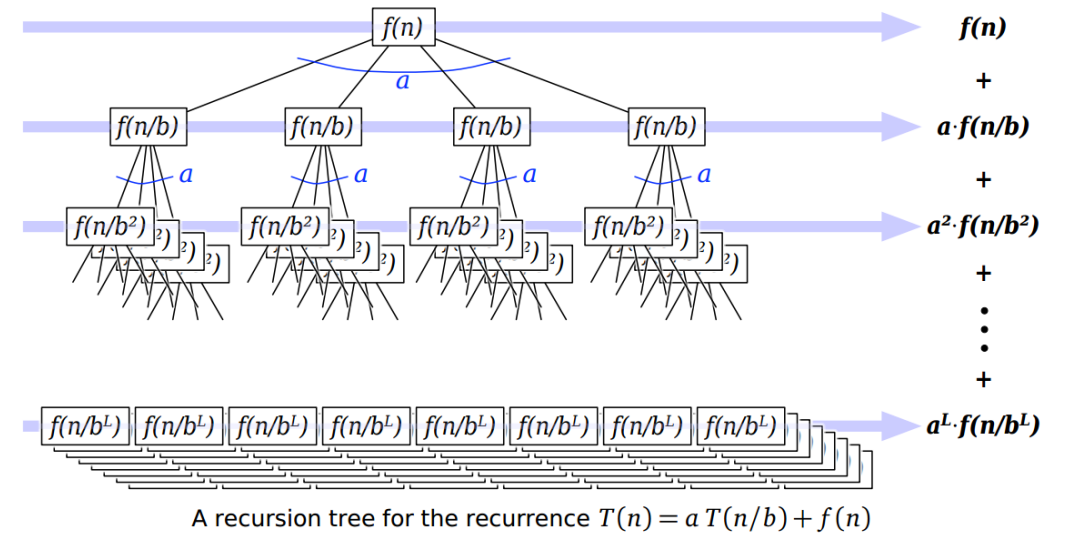
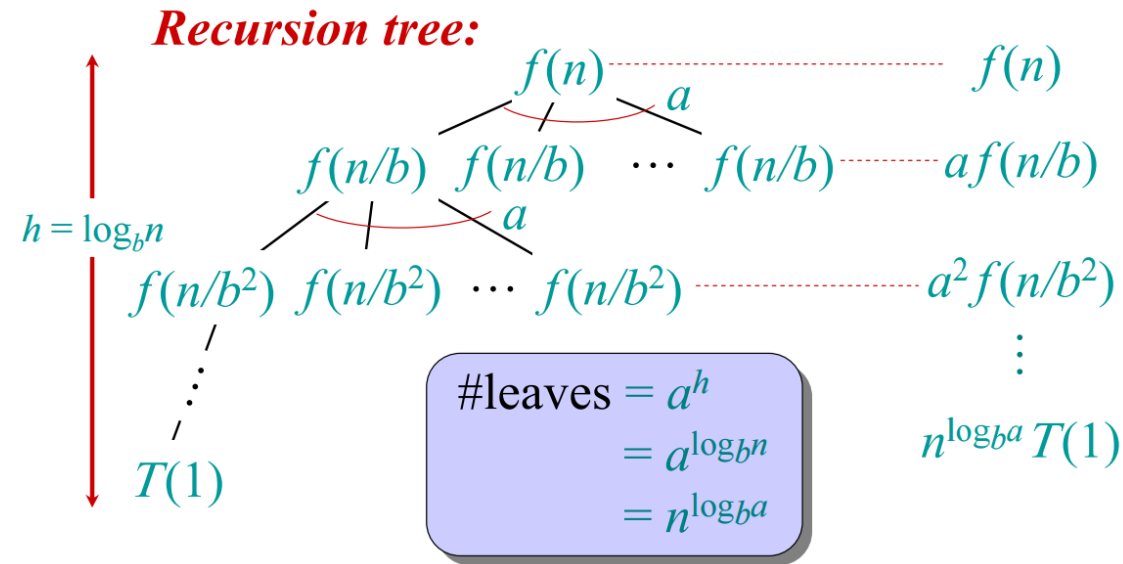
Master Method: Interpretation

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



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

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

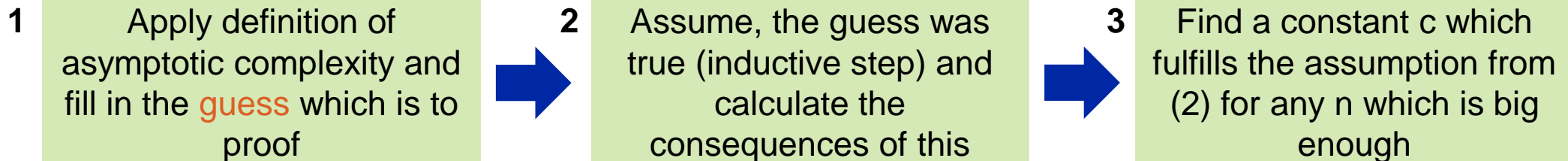
Task 1.4b from midterm 1 of FS 2016:

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:

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

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

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

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

$$10 \leq c$$



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



Preview on Exercise 4

- Task 1: Finding the Odd Element Efficiently
- Task 2: Maximum-Subarray Algorithm
- Tasks 3 and 4: Recurrences

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.

- 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.
- Provide a C code for divide and conquer odd occurring element finder algorithm.

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 – Tasks 3 and 4: Recurrences



**Universität
Zürich** ^{UZH}

Institut für Informatik

Wrap-Up

- Summary
- Outlook
- Questions



Outlook

Next tutorial: Friday, 20.03.2020, Y35-F-32 (unless other information will be provided)

Topics:

- Review of Exercise 4
- Recurrences
- Divide and Conquer
- Preparation for Midterm 1
- Preview to Exercise 5
- ... (your wishes)

Midterm 1: Monday, 23.03.2020, 12:15 – 13:45 h



Questions?



Thank you for your attention.