

Data Structures and Algorithms

Chapter 2

Werner Nutt

Acknowledgments

- The course follows the book “Introduction to Algorithms”, by **Cormen, Leiserson, Rivest and Stein**, MIT Press [CLRST]. Many examples displayed in these slides are taken from their book.
- These slides are based on those developed by Michael Böhlen for this course.

(See <http://www.inf.unibz.it/dis/teaching/DSA/>)

- The slides also include a number of additions made by Roberto Sebastiani and Kurt Ranalter when they taught later editions of this course

(See http://disi.unitn.it/~rseba/DIDATTICA/dsa2011_BZ//)

DSA, Chapter 2: Overview

- Complexity of algorithms
- Asymptotic analysis
- Correctness of algorithms
- Special case analysis

DSA, Chapter 2: Overview

- Complexity of algorithms
- Asymptotic analysis
- Special case analysis
- Correctness of algorithms

Analysis of Algorithms

- Efficiency:
 - Running time
 - Space used
- Efficiency is defined as a function of the input size:
 - **Number of data elements** (numbers, points)
 - The **number of bits** of an input number

The RAM Model

We study complexity on a **simplified machine model**, the **RAM** (= Random Access Machine):

- accessing and manipulating data takes a (small) constant amount of time

Among the **instructions** (each taking constant time), we usually choose one type of instruction as a **characteristic operation** that is counted:

- arithmetic (add, subtract, multiply, etc.)
- data movement (assign)
- control flow (branch, subroutine call, return)
- comparison

Data types: integers, characters, and floats

Analysis of Insertion Sort

Running time as a function of the input size
(exact analysis)

	cost	times
for $j := 2$ to n do	c1	n
$key := A[j]$	c2	$n-1$
// Insert $A[j]$ into $A[1..j-1]$		
$i := j-1$	c3	$n-1$
while $i > 0$ and $A[i] > key$ do	c4	$\sum_{j=2}^n t_j$
$A[i+1] := A[i]$	c5	$\sum_{j=2}^n (t_j - 1)$
$i--$	c6	$\sum_{j=2}^n (t_j - 1)$
$A[i+1] := key$	c7	$n-1$

t_j is the number of times the while loop is executed, i.e.,

$(t_j - 1)$ is number of elements in the initial segment greater than $A[j]$

Analysis of Insertion Sort/2

- The running time of an algorithm for a given input is the sum of the running times of each statement.
- A statement
 - with cost c
 - that is executed n timescontributes $c*n$ to the running time.
- The total running time $T(n)$ of insertion sort is

$$T(n) = c1*n + c2*(n-1) + c3*(n-1) + c4 * \sum_{j=2}^n t_j \\ + c5 \sum_{j=2}^n (t_j - 1) + c6 \sum_{j=2}^n (t_j - 1) + c7*(n - 1)$$

Analysis of Insertion Sort/3

- The running time is not necessarily equal for every input of size n
- The performance depends on the details of the input (not only length n)
- This is modeled by t_j
- In the case of Insertion Sort, the time t_j depends on the original sorting of the input array

Performance Analysis

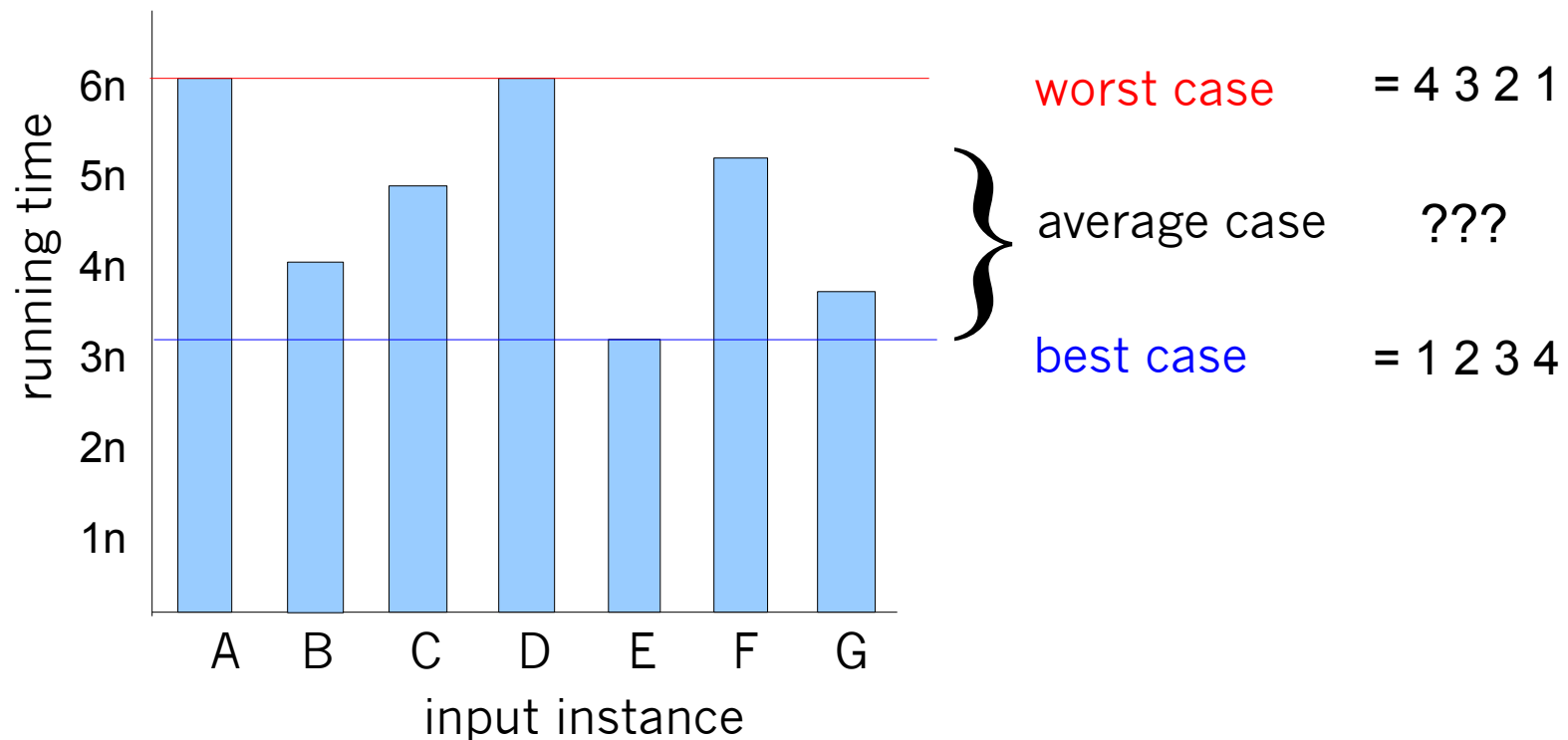
- Often it is sufficient to count the number of iterations of the core (innermost) part
 - no distinction between comparisons, assignments, etc (that means, roughly the same cost for all of them)
 - gives precise enough results
- In some cases the cost of selected operations dominates all other costs.
 - disk I/O versus RAM operations
 - database systems

Worst/Average/Best Case

- Analyzing Insertion Sort's
 - **Worst case:** elements sorted in inverse order, $t_j=j$, total running time is *quadratic* (time = an^2+bn+c)
 - **Average case (= average of all inputs of size n):** $t_j=j/2$, total running time is *quadratic* (time = an^2+bn+c)
 - **Best case:** elements already sorted, $t_j=1$, innermost loop is never executed, total running time is *linear* (time = $an+b$)
- How can we define these concepts formally?
 - ... and how much sense does “best case” make?

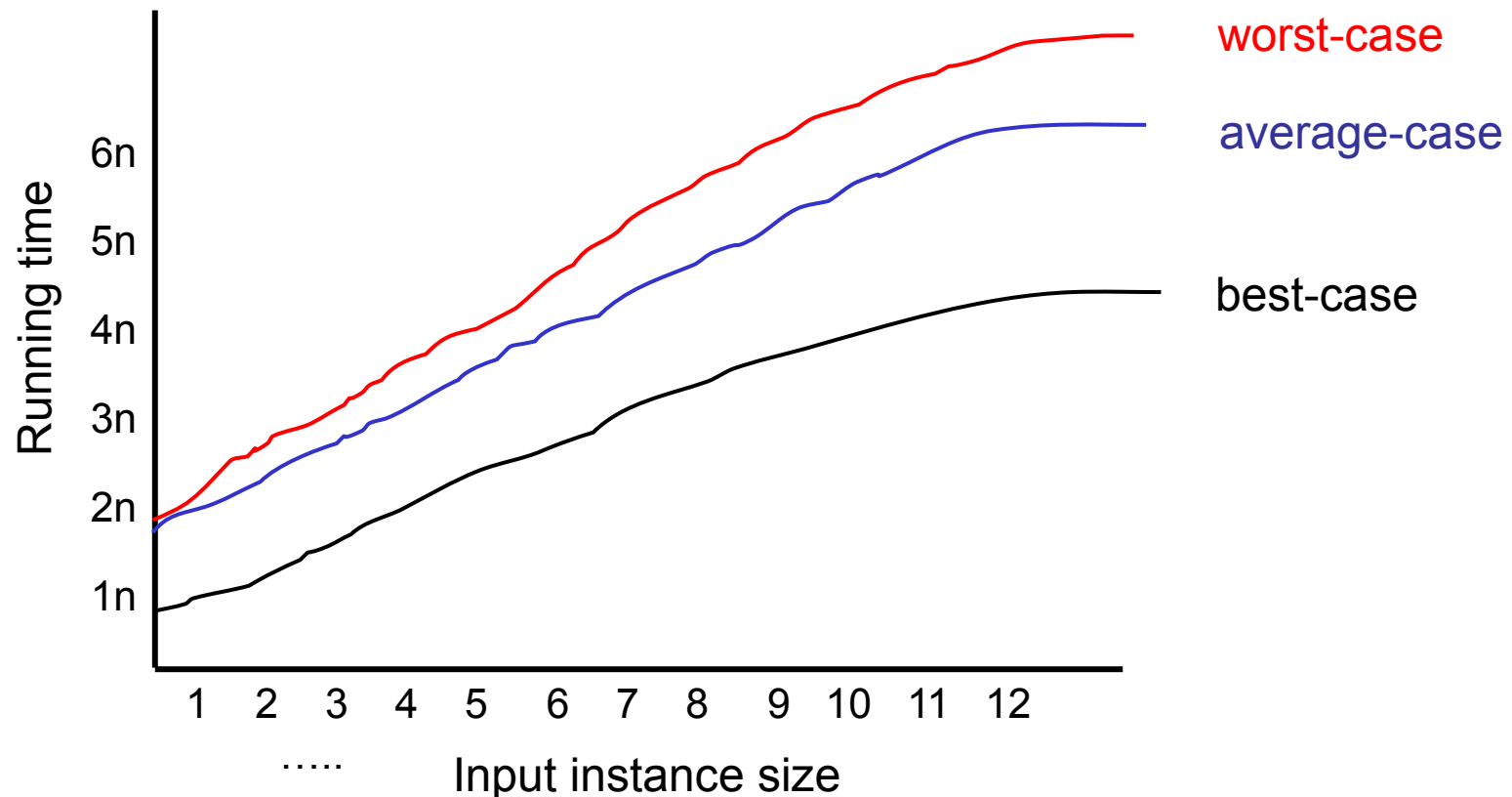
Worst/Average/Best Case/2

For a specific size of input size n , investigate running times for different input instances:



Worst/Average/Best Case/3

For inputs of all sizes:



Best/Worst/Average Case/4

Worst case is most often used:

- It is an upper-bound
- In certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
- For some algorithms, **worst case** occurs fairly often
- The **average case** is often as bad as the **worst case**

The **average case** depends on assumptions

- What are the possible input cases?
- What is the probability of each input?

Analysis of Linear Search

INPUT: $A[1..n]$ – an array of integers,
 q – an integer.

OUTPUT: j s.t. $A[j]=q$, or -1 if $\forall j(1 \leq j \leq n): A[j] \neq q$

$j := 1$

while $j \leq n$ **and** $A[j] \neq q$ **do** $j++$

if $j \leq n$ **then return** j

else return -1

- Worst case running time: n
- Average case running time: $(n+1)/2$ (if q is present)
... under which assumption?

Binary Search: Idea

- Search in a **sorted array**
- Check the element in the middle of the array
- If we have found the search value, we are done
- If not, check whether the search value has to be in the left or in the right half of the array
- Depending on the check, continue with the left or the right half ...

Binary Search, Recursive Version

INPUT: $A[1..n]$ – sorted (increasing) array of integers, q – integer.

OUTPUT: an index j such that $A[j] = q$. -1, if $\forall j (1 \leq j \leq n): A[j] \neq q$

```
searchRec (A, q)
  searchRecAux (A, q, 1, n)
```

```
searchRecAux (A, q, l, r)
  m :=  $\lfloor (l+r)/2 \rfloor$  ;
  if l > r
    then return -1
  else if A(m) = q
    then return m
  else if q < A(m)
    then return searchRecAux (A, q, l, m-1)
  else return searchRecAux (A, q, m+1, r)
```

Binary Search, Iterative Version

INPUT: $A[1..n]$ – sorted (increasing) array of integers, q – integer.

OUTPUT: an index j such that $A[j] = q$. -1, if $\forall j (1 \leq j \leq n): A[j] \neq q$

```
searchIter(A, q)
  l := 1; r := n;
  m :=  $\lfloor (l+r)/2 \rfloor$ ;
  while l ≤ r and A(m) ≠ q do
    if q < A(m)
      then r := m-1
    else l := m+1
  m :=  $\lfloor (l+r)/2 \rfloor$ ;
  if l > r
    then return -1
    else return m
```

Analysis of Binary Search

How many times is the loop executed?

- With each execution the difference between l and r is cut in half
 - Initially the difference is n
 - *The loop stops when the difference becomes 0 (less than 1)*
- How many times do you have to cut n in half to get 0?
- $\log n$ – better than the brute-force approach of linear search (n).

Linear vs Binary Search

- Costs of linear search: n
- Costs of binary search: $\log(n)$
- Should we care?
- Phone book with n entries:
 - $n = 200,000$, $\log n = \log 200,000 = 8 + 10$
 - $n = 2M$, $\log 2M = 1 + 10 + 10$
 - $n = 20M$, $\log 20M = 5 + 20$

DSA, Part 2: Overview

- Complexity of algorithms
- **Asymptotic analysis**
- Special case analysis
- Correctness of algorithms

Asymptotic Analysis

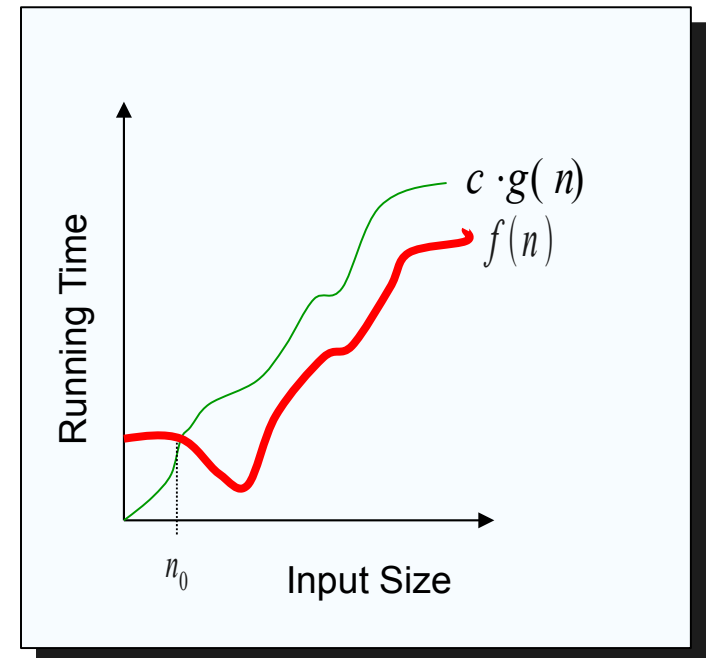
- Goal: simplify the analysis of the running time by getting rid of details, which are affected by specific implementation and hardware
 - “rounding” of numbers: $1,000,001 \approx 1,000,000$
 - “rounding” of functions: $3n^2 \approx n^2$
- Capturing the essence: how the **running time** of an algorithm increases with the size of the input *in the limit*
 - Asymptotically more efficient algorithms are **best for all but small inputs**

Asymptotic Notation

The “big-Oh” O-Notation

- talks about asymptotic upper bounds
- $f(n) = O(g(n))$ iff there exist $c > 0$ and $n_0 > 0$, s.t. $f(n) \leq c g(n)$ for $n \geq n_0$
- $f(n)$ and $g(n)$ are functions over non-negative integers

Used for *worst-case* analysis



Asymptotic Notation, Example

$$f(n) = 2n^2 + 3(n+1), \quad g(n) = 3n^2$$

Values of $f(n) = 2n^2 + 3(n+1)$:

$$2+6, \quad 8+9, \quad 18+12, \quad 32+15$$

Values of $g(n) = 3n^2$:

$$3, \quad 12, \quad 27, \quad 64$$

From $n_0 = 4$ onward, we have $f(n) \leq g(n)$

Asymptotic Notation, Examples

- Simple Rule: We can always drop lower order terms and constant factors, without changing big Oh:

$$- 7n + 3 \quad \text{is} \quad O(n)$$

$$- 8n^2 \log n + 5n^2 + n \quad \text{is} \quad O(n^2 \log n)$$

$$- 50 n \log n \quad \text{is} \quad O(n \log n)$$

- Note:

$$- 50 n \log n \quad \text{is} \quad O(n^2)$$

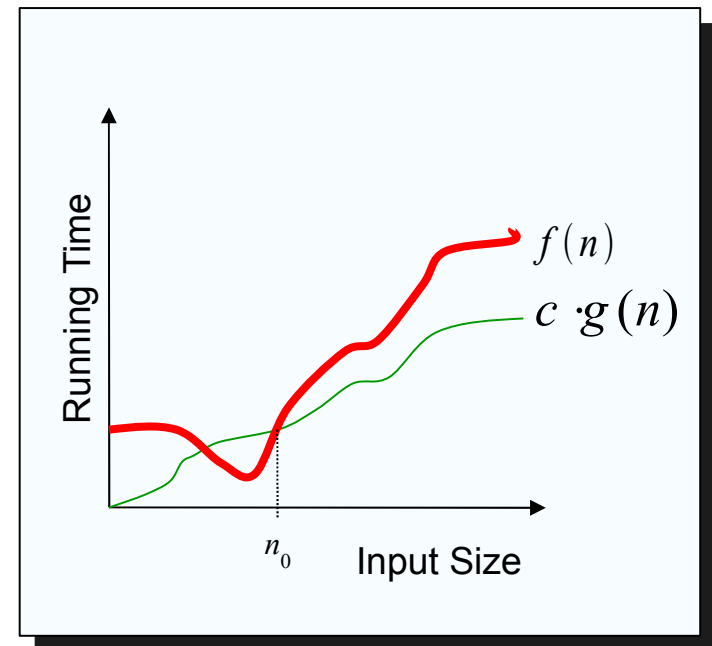
$$- 50 n \log n \quad \text{is} \quad O(n^{100})$$

but this is less informative than saying

$$- 50 n \log n \quad \text{is} \quad O(n \log n)$$

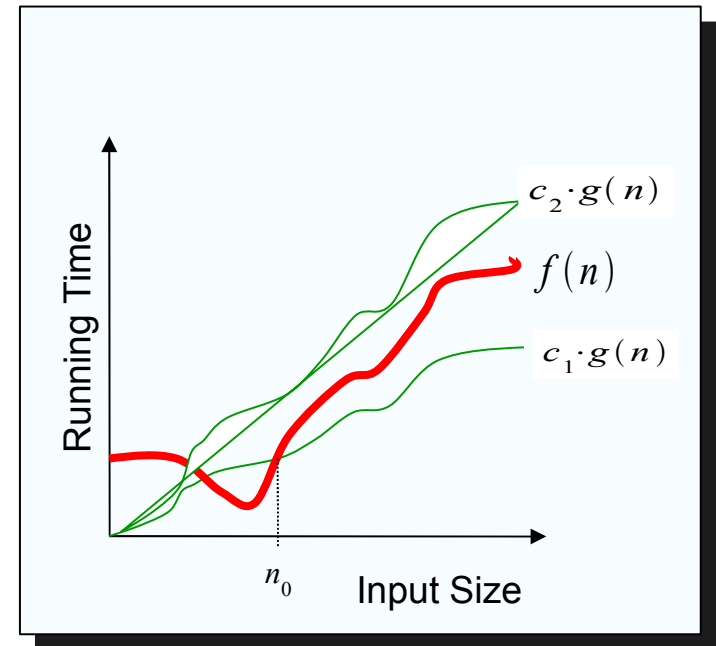
Asymptotic Notation/2

- The “big-Omega” Ω -Notation
 - asymptotic lower bound
 - $f(n) = \Omega(g(n))$ iff
there exist $c > 0$ and $n_0 > 0$,
s.t. $c g(n) \leq f(n)$, for $n \geq n_0$
- Used to describe lower bounds of algorithmic problems
 - E.g., searching in
a sorted array
with linear search is $\Omega(n)$,
with binary search is $\Omega(\log n)$



Asymptotic Notation/3

- The “big-Theta” Θ -Notation
 - asymptotically tight bound
 - $f(n) = \Theta(g(n))$ if there exists $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$,
s.t. for $n \geq n_0$
$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$
- $f(n) = \Theta(g(n))$ iff
 $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- *Note:* $O(f(n))$ is often used
when $\Theta(f(n))$ is meant



Asymptotic Notation/4

- Analogy with real numbers
 - $f(n) = O(g(n)) \cong f \leq g$
 - $f(n) = \Omega(g(n)) \cong f \geq g$
 - $f(n) = \Theta(g(n)) \cong f = g$

- Abuse of notation:
 - $f(n) = O(g(n))$ actually means
 $f(n) \in O(g(n))$

Exercise: Asymptotic Growth

Order the following functions according to their asymptotic growth.

- $2^n + n^2$
- $3n^3 + n^2 - 2n^3 + 5n - n^3$
- $20 \log_2 2n$
- $20 \log_2 n^2$
- $20 \log_2 4^n$
- $20 \log_2 2^n$
- 3^n

Comparison of Running Times

Determining the maximal problem size

Running Time $T(n)$ in μs	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$20n \log n$	4,096	166,666	7,826,087
$2n^2$	707	5,477	42,426
n^4	31	88	244
2^n	19	25	31

DSA, Part 2: Overview

- Complexity of algorithms
- Asymptotic analysis
- **Special case analysis**
- Correctness of algorithms

Special Case Analysis

- Consider extreme cases and make sure your solution works in all cases.
- The problem: identify special cases.
- This is related to INPUT and OUTPUT specifications.

Special Cases

- empty data structure (array, file, list, ...)
- single element data structure
- completely filled data structure
- entering a function
- termination of a function
- zero, empty string
- negative number
- border of domain
- start of loop
- end of loop
- first iteration of loop

Sortedness

The following algorithm checks whether an array is sorted.

INPUT: $A[1..n]$ - an array of integers.

OUTPUT: TRUE if A is sorted; FALSE otherwise

```
for  $i := 1$  to  $n$   
  if  $A[i] \geq A[i+1]$  then return FALSE  
return TRUE
```

Analyze the algorithm by considering special cases.

Sortedness/2

INPUT: $A[1..n]$ - an array of integers.

OUTPUT: TRUE if A is sorted; FALSE otherwise

```
for  $i := 1$  to  $n$   
  if  $A[i] \geq A[i+1]$  then return FALSE  
return TRUE
```

- Start of loop, $i=1 \rightarrow$ OK
- End of loop, $i=n \rightarrow$ ERROR (tries to access $A[n+1]$)

Sortedness/3

INPUT: $A[1..n]$ - an array of integers.

OUTPUT: TRUE if A is sorted; FALSE otherwise

```
for  $i := 1$  to  $n-1$   
  if  $A[i] \geq A[i+1]$  then return FALSE  
return TRUE
```

- Start of loop, $i=1$ ⊙ OK
- End of loop, $i=n-1$ ⊙ OK
- $A=[1,2,3]$ ⊙ First iteration, from $i=1$ to $i=2$ ⊙ OK
- $A=[1,2,2]$ ⊙ **ERROR** (if $A[i]=A[i+1]$ for some i)

Sortedness/4

INPUT: $A[1..n]$ - an array of integers.

OUTPUT: TRUE if A is sorted; FALSE otherwise

```
for  $i := 1$  to  $n-1$   
  if  $A[i] > A[i+1]$  then return FALSE  
return TRUE
```

- Start of loop, $i=1 \rightarrow$ OK
- End of loop, $i=n-1 \rightarrow$ OK
- $A=[1,2,3] \rightarrow$ First iteration, from $i=1$ to $i=2 \rightarrow$ OK
- $A=[1,1,1] \rightarrow$ OK
- Empty data structure, $n=0 \rightarrow ?$ (for loop)
- $A=[-1,0,1,-3] \rightarrow$ OK

Binary Search, Variant 1

Analyze the following algorithm by considering special cases.

```
l := 1; r := n
do
  m :=  $\lfloor (l+r)/2 \rfloor$ 
  if A[m] = q then return m
  else if A[m] > q then r := m-1
  else l := m+1
while l < r
return -1
```

Binary Search, Variant 1

```
l := 1; r := n
do
  m :=  $\lfloor (l+r)/2 \rfloor$ 
  if A[m] = q then return m
  else if A[m] > q then r := m-1
  else l := m+1
while l < r
return -1
```

- Start of loop → OK
- End of loop, $l=r$ → **Error! Example: search 3 in [3 5 7]**

Binary Search, Variant 1

```
l := 1; r := n
do
  m :=  $\lfloor (l+r)/2 \rfloor$ 
  if A[m] = q then return m
  else if A[m] > q then r := m-1
  else l := m+1
while l <= r
return -1
```

- Start of loop → OK
- End of loop, $l=r$ → OK
- First iteration → OK
- $A=[1,1,1]$ → OK
- Empty array, $n=0$ → **Error! Tries to access $A[0]$**
- One-element array, $n=1$ → OK

Binary Search, Variant 1

```
l := 1; r := n
if r < l then return -1;
do
  m :=  $\lfloor (l+r)/2 \rfloor$ 
  if A[m] = q then return m
  else if A[m] > q then r := m-1
  else l := m+1
while l <= r
return -1
```

- Start of loop → OK
- End of loop, l=r → OK
- First iteration → OK
- A=[1,1,1] → OK
- Empty data structure, n=0 → OK
- One-element data structure, n=1 → OK

Binary Search, Variant 2

Analyze the following algorithm by considering special cases

```
l := 1; r := n
while l < r do
  m :=  $\lfloor (l+r)/2 \rfloor$ 
  if A[m] <= q
    then l := m+1 else r := m
if A[l-1] = q
  then return l-1 else return -1
```

Binary Search, Variant 3

Analyze the following algorithm by considering special cases

```
l := 1; r := n
while l <= r do
  m :=  $\lfloor (l+r)/2 \rfloor$ 
  if A[m] <= q
    then l := m+1 else r := m
if A[l-1] = q
  then return l-1 else return -1
```

Insertion Sort, Slight Variant

- Analyze the following algorithm by considering special cases
- Hint: beware of lazy evaluations

INPUT: $A[1..n]$ - an array of integers

OUTPUT: permutation of A s.t.

$A[1] \leq A[2] \leq \dots \leq A[n]$

for $j := 2$ **to** n **do**

$key := A[j]; i := j-1;$

while $A[i] > key$ **and** $i > 0$ **do**

$A[i+1] := A[i]; i--;$

$A[i+1] := key$

Merge

Analyze the following algorithm
by considering special cases.

INPUT: $A[1..n_1]$, $B[1..n_2]$ sorted arrays of
integers, $C[1..n_1+n_2]$ array

OUTPUT: permutation C of $A.B$ s.t.
 $C[1] \leq C[2] \leq \dots \leq C[n_1+n_2]$

```
i:=1; j:=1;
for k:=1 to n1 + n2 do
    if A[i] <= B[j]
    then C[k] := A[i]; i++;
    else C[k] := B[j]; j++;
return C;
```

Merge/2

INPUT: $A[1..n_1]$, $B[1..n_2]$ sorted arrays of integers, $C[1..n_1+n_2]$ array

OUTPUT: permutation C of $A.B$ s.t.

$C[1] \leq C[2] \leq \dots \leq C[n_1+n_2]$

$i := 1 ; j := 1 ;$

for $k := 1$ **to** $n_1 + n_2$ **do**

if $j > n_2$ **or** $(i \leq n_1$ **and** $A[i] \leq B[j])$

then $C[k] := A[i]; i++;$

else $C[k] := B[j]; j++;$

return $C;$

Merge/3

To analyze the algorithm on the previous slide, we distinguish 4 cases

- neither A nor B exhausted implies “ $A[i] \leq B[j]$ ” decides
- B exhausted, A not, implies $j > n_2$, implies A wins
- B not exhausted, A exhausted, implies $j \leq n_2 \ \&\& \ i > n_1$, implies B wins
- A, B both exhausted, implies $k > n_1 + n_2$, implies algorithm stops

DSA, Part 2: Overview

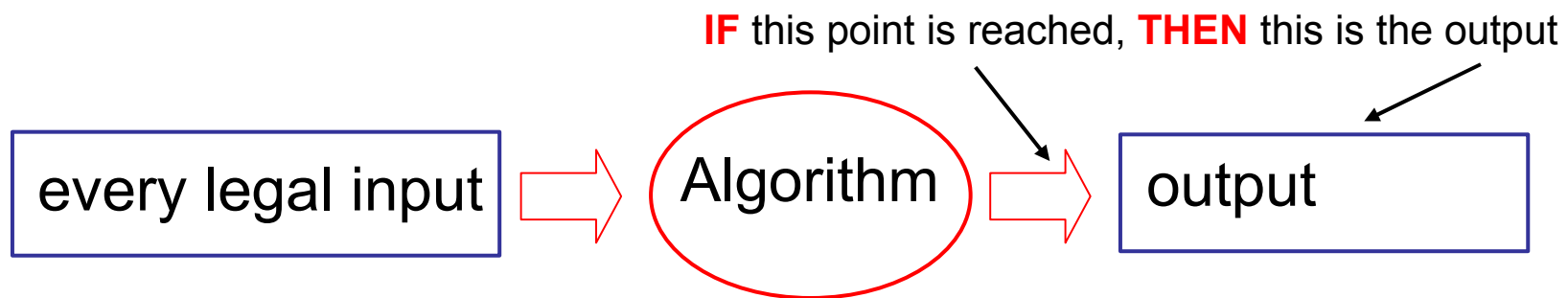
- Complexity of algorithms
- Asymptotic analysis
- Special case analysis
- **Correctness of algorithms**

Correctness of Algorithms

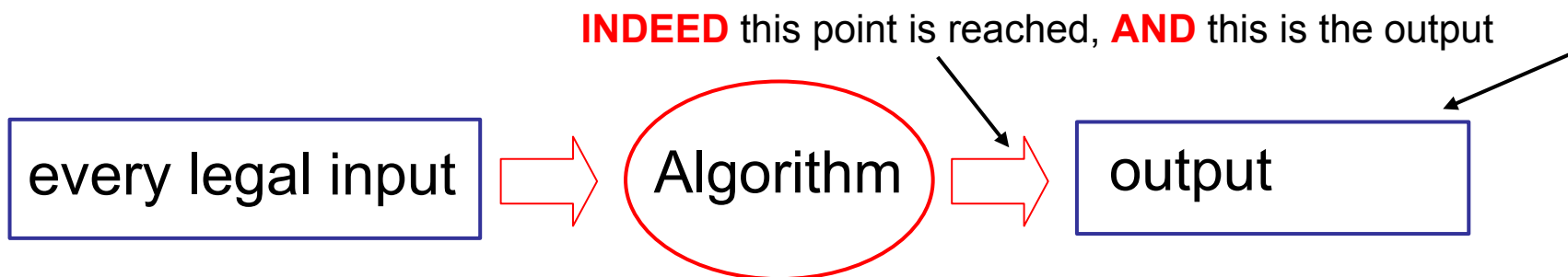
- An algorithm is *correct* if for every legal input, it terminates and produces the desired output.
- Automatic proof of correctness is not possible (this is one of the so-called “undecidable problems”)
- There are **practical techniques** and **rigorous formalisms** that help one to reason about the correctness of (parts of) algorithms.

Partial and Total Correctness

- Partial correctness



- Total correctness



Assertions

- To prove partial correctness we associate a number of **assertions** (statements about the state of the execution) with specific **checkpoints** in the algorithm.
 - E.g., “ $A[1], \dots, A[j]$ form an increasing sequence”
- **Preconditions** – assertions that must be valid *before* the execution of an algorithm or a subroutine (**INPUT**)
- **Postconditions** – assertions that must be valid *after* the execution of an algorithm or a subroutine (**OUTPUT**)

Pre- and Postconditions of Linear Search

INPUT: $A[1..n]$ - an array of integers,
 q - an integer.

OUTPUT: j s.t. $A[j]=q$. -1 if $\forall i(1 \leq i \leq n): A[i] \neq q$

$j := 1$

while $j \leq n$ **and** $A[j] \neq q$ **do** $j++$

if $j \leq n$ **then return** j

else return -1

How can we be sure that

- whenever the precondition holds,
- also the postcondition holds?

Loop Invariant in Linear Search

```
j := 1
while j ≤ n and A[j] ≠ q do j++
if j ≤ n then return j
else return -1
```

Whenever the beginning of the loop is reached, then

$$A[i] \neq q \quad \text{for all } i \text{ where } 1 \leq i < j$$

When the loop stops, there are two cases

- $j = n+1$, which implies $A[i] \neq q$ for all i , $1 \leq i < n+1$
- $A[j] = q$

Loop Invariant in Linear Search

```
j := 1
while j ≤ n and A[j] ≠ q do j++
if j ≤ n then return j
else return -1
```

Note: The condition

$A[i] \neq q$ for all i where $1 \leq i < j$

- holds when the loop is entered for the first time
- continues to hold until we reach the loop for the last time

Loop Invariants

- **Invariants:** assertions that are valid every time the **beginning of the loop** is reached (many times during the execution of an algorithm)
- We must show three things about loop invariants:
 - **Initialization:** it is true prior to the first iteration.
 - **Maintenance:** *if* it is true before an iteration, *then* it is true after the iteration.
 - **Termination:** when a loop terminates, the invariant gives a useful property to show the correctness of the algorithm

Example: Binary Search/1

- We want to show that q is not in A
if -1 is returned.

- Invariant:**

$\forall i \in [1..l-1]: A[i] < q$ (ia)

$\forall i \in [r+1..n]: A[i] > q$ (ib)

- Initialization:** $l = 1, r = n$

the invariant holds because

there are no elements to the left of l or to the right of r .

$l = 1$ yields $\forall i \in [1..0]: A[i] < q$

this holds because $[1..0]$ is empty

$r = n$ yields $\forall i \in [n+1..n]: A[i] > q$

this holds because $[n+1..n]$ is empty

```

l := 1; r := n;
m := ⌊(l+r)/2⌋;
while l ≤ r and A(m) ≠ q do
  if q < A(m)
    then r := m-1
    else l := m+1
  m := ⌊(l+r)/2⌋;
if l > r
  then return -1
else return m

```


Example: Binary Search/2

- **Invariant:**

$\forall i \in [1..l-1]: A[i] < q$ (ia)

$\forall i \in [r+1..n]: A[i] > q$ (ib)

```

l := 1; r := n;
m := ⌊(l+r)/2⌋;
while l ≤ r and A(m) ≠ q do
    if q < A(m)
        then r := m-1
        else l := m+1
    m := ⌊(l+r)/2⌋;
if l > r
    then return -1
    else return m

```

- **Maintenance:** $1 \leq l, r \leq n, m = \lfloor (l+r)/2 \rfloor$

We consider two cases:

- $A[m] \neq q$ & $q < A[m]$: implies $r = m-1$
 A sorted implies $\forall k \in [r+1..n]: A[k] > q$ (ib)
- $A[m] \neq q$ & $A[m] < q$: implies $l = m+1$
 A sorted implies $\forall k \in [1..l-1]: A[k] < q$ (ia)

Example: Binary Search/3

- **Invariant:**

$\forall i \in [1..l-1]: A[i] < q$ (ia)

$\forall i \in [r+1..n]: A[i] > q$ (ib)

```

l := 1; r := n;
m := ⌊(l+r)/2⌋;
while l ≤ r and A(m) ≠ q do
    if q < A(m)
        then r := m-1
        else l := m+1
    m := ⌊(l+r)/2⌋;
if l > r
    then return -1
    else return m
  
```

- **Termination:** $1 \leq l, r \leq n, l \leq r$

Two cases:

$l := m+1$ implies $l_{new} = \lfloor (l+r)/2 \rfloor + 1 > l_{old}$

$r := m-1$ implies $r_{new} = \lfloor (l+r)/2 \rfloor - 1 < r_{old}$

- The range gets smaller during each iteration and the loop will terminate when $l \leq r$ no longer holds

Example: Insertion Sort/1

Loop invariants:

External “for” loop

Let A^{orig} denote the array at the beginning of the for loop:

$A[1..j-1]$ is sorted

$A[1..j-1] \in A^{\text{orig}}[1..j-1]$

Internal “while” loop

Let A^{orig} denote the array at beginning of the while loop:

- $A[1..i] = A^{\text{orig}}[1..i]$
- $A[i+2..j] = A^{\text{orig}}[i+1..j-1]$
- $A[k] > \text{key}$ for all k in $\{i+2, \dots, j\}$

```

for j := 2 to n do
  key := A[j]
  i := j-1
  while i>0 and A[i]>key do
    A[i+1] := A[i]
    i--
  A[i+1] := key

```

Example: Insertion Sort/2

External for loop:

- (i) $A[1..j-1]$ is sorted
- (ii) $A[1..j-1] \in A^{\text{orig}}[1..j-1]$

Internal while loop:

- $A[1..i] = A^{\text{orig}}[1..i]$
- $A[i+2..j] = A^{\text{orig}}[i+1..j-1]$
- $A[k] > \text{key}$ for all k in $\{i+2, \dots, j\}$

```

for j := 2 to n do
  key := A[j]
  i := j-1
  while i>0 and A[i]>key do
    A[i+1] := A[i]
    i--
  A[i+1] := key

```

Initialization:

External loop: (i), (ii) $j = 2$: $A[1..1] \in A^{\text{orig}}[1..1]$ and is trivially sorted

Internal loop: $i = j-1$:

- $A[1..j-1] = A^{\text{orig}}[1..j-1]$, since nothing has happened
- $A[j+1..j] = A^{\text{orig}}[j..j-1]$, since both sides are empty
- $A[k] > \text{key}$ holds trivially for all k in the empty set

Example: Insertion Sort/3

External for loop:

- (i) $A[1..j-1]$ is sorted
- (ii) $A[1..j-1] \in A^{\text{orig}}[1..j-1]$

Internal while loop:

- $A[1..i] = A^{\text{orig}}[1..i]$
- $A[i+2..j] = A^{\text{orig}}[i+1..j-1]$
- $A[k] > \text{key}$ for all k in $\{i+2, \dots, j\}$

```

for j := 2 to n do
  key := A[j]
  i := j-1
  while i > 0 and A[i] > key do
    A[i+1] := A[i]
    i--
  A[i+1] := key

```

Maintenance internal while loop

Before the decrement “ $i--$ ”, the following facts hold:

- $A[1..i-1] = A^{\text{orig}}[1..i-1]$ (because nothing in $A[1..i-1]$ has been changed)
- $A[i+1..j] = A^{\text{orig}}[i..j-1]$ (because $A[i]$ has been copied to $A[i+1]$ and
 $A[i+2..j] = A^{\text{orig}}[i+1..j-1]$)
- $A[k] > \text{key}$ for all k in $\{i+1, \dots, j\}$ (because $A[i]$ has been copied to $A[i+1]$)

After the decrement “ $i--$ ”, the invariant holds because $i-1$ is replaced by i .

Example: Insertion Sort/4

External for loop:

- (i) $A[1..j-1]$ is sorted
- (ii) $A[1..j-1] \in A^{\text{orig}}[1..j-1]$

Internal while loop:

- $A[1..i] = A^{\text{orig}}[1..i]$
- $A[i+2..j] = A^{\text{orig}}[i+1..j-1]$
- $\text{key} < A[k]$ for all k in $\{i+2, \dots, j\}$

```

for j := 2 to n do
  key := A[j]
  i := j-1
  while i>0 and A[i]>key do
    A[i+1] := A[i]
    i--
  A[i+1] := key

```

Termination internal while loop

The while loop terminates, since i is decremented in each round.

Termination can be due to two reasons:

$i=0$: $A[2..j] = A^{\text{orig}}[1..j-1]$ and $\text{key} < A[k]$ for all k in $\{2, \dots, j\}$ (because of the invariant)
 implies $\text{key}, A[2..j]$ is a sorted version of $A^{\text{orig}}[1..j]$

$A[i] \leq \text{key}$: $A[1..i] = A^{\text{orig}}[1..i]$, $A[i+2..j] = A^{\text{orig}}[i+1..j-1]$, $\text{key} = A^{\text{orig}}[j]$
 implies $A[1..i], \text{key}, A[i+2..j]$ is a sorted version of $A^{\text{orig}}[1..j]$

Example: Insertion Sort/5

External for loop:

- (i) $A[1..j-1]$ is sorted
- (ii) $A[1..j-1] \in A^{\text{orig}}[1..j-1]$

Internal while loop:

- $A[1..i] = A^{\text{orig}}[1..i]$
- $A[i+2..j] = A^{\text{orig}}[i+1..j-1]$
- $\text{key} < A[k]$ for all k in $\{i+2, \dots, j\}$

```

for j := 2 to n do
  key := A[j]
  i := j-1
  while i > 0 and A[i] > key do
    A[i+1] := A[i]
    i--
  A[i+1] := key

```

Maintenance of external for loop

When the internal while loop terminates, we have (see previous slide):

$A[1..i], \text{key}, A[i+2..j]$ is a sorted version of $A^{\text{orig}}[1..j]$

After

- assigning key to $A[i+1]$ and
- Incrementing j ,

the invariant of the external loop holds again.

Example: Insertion Sort/6

External for loop:

- (i) $A[1..j-1]$ is sorted
- (ii) $A[1..j-1] \in A^{\text{orig}}[1..j-1]$

Internal while loop:

- $A[1..i] = A^{\text{orig}}[1..i]$
- $A[i+2..j] = A^{\text{orig}}[i+1..j-1]$
- $\text{key} < A[k]$ for all k in $\{i+2, \dots, j\}$

```

for j := 2 to n do
  key := A[j]
  i := j-1
  while i>0 and A[i]>key do
    A[i+1] := A[i]
    i--
  A[i+1] := key

```

Termination of external for loop

The for loop terminates because j is incremented in each round.

Upon termination, $j = n+1$ holds.

In this situation, the loop invariant of the for loop says:

$A[1..n]$ is sorted and contains the same values as $A^{\text{orig}}[1..n]$

That is, A has been sorted.

Example: Bubble Sort

INPUT: $A[1..n]$ – an array of integers

OUTPUT: permutation of A s.t. $A[1] \leq A[2] \leq \dots \leq A[n]$

```
for j := 1 to n-1 do
  for i := n downto j+1 do
    if  $A[i-1] > A[i]$  then
      swap( $A, i-1, i$ )
```

- What is a good loop invariant for the outer loop?
(i.e., a property that always holds at the end of line 1)
- ... and what is a good loop invariant for the inner loop?
(i.e., a property that always holds at the end of line 2)

Exercise

- Apply the same approach that we used for insertion sort to prove the correctness of bubble sort and selection sort.

Math Refresher

- Arithmetic progression

$$\sum_{i=0}^n i = 0 + 1 + \dots + n = n(n+1)/2$$

- Geometric progression (for a number $a \neq 1$)

$$\sum_{i=0}^n a^i = 1 + a^2 + \dots + a^n = (1 - a^{n+1}) / (1 - a)$$

Induction Principle

We want to show that property P is true for all integers $n \geq n_0$.

Basis: prove that P is true for n_0 .

Inductive step: prove that if P is true for all k such that $n_0 \leq k \leq n - 1$ then P is also true for n .

Exercise: Prove that every Fibonacci number of the form $\text{fib}(3n)$ is even

Summary

- Algorithmic complexity
- Asymptotic analysis
 - Big O and Theta notation
 - Growth of functions and asymptotic notation
- Correctness of algorithms
 - Pre/Post conditions
 - Invariants
- Special case analysis