Data Structures and Algorithms

Part 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, Part 2: Overview

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

DSA, Part 2: Overview

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

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

It is important to choose the level of detail.

The RAM (Random Access Machine) model:

- 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]                 0      n-1
  i := j-1                                      c3     n-1
  while i>0 and A[i]>key do                     c4     ∑_{j=2}^{n} t_j
    A[i+1] := A[i]                              c5     ∑_{j=2}^{n} (t_j-1)
    i--                                         c6     ∑_{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 times

  contributes 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 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

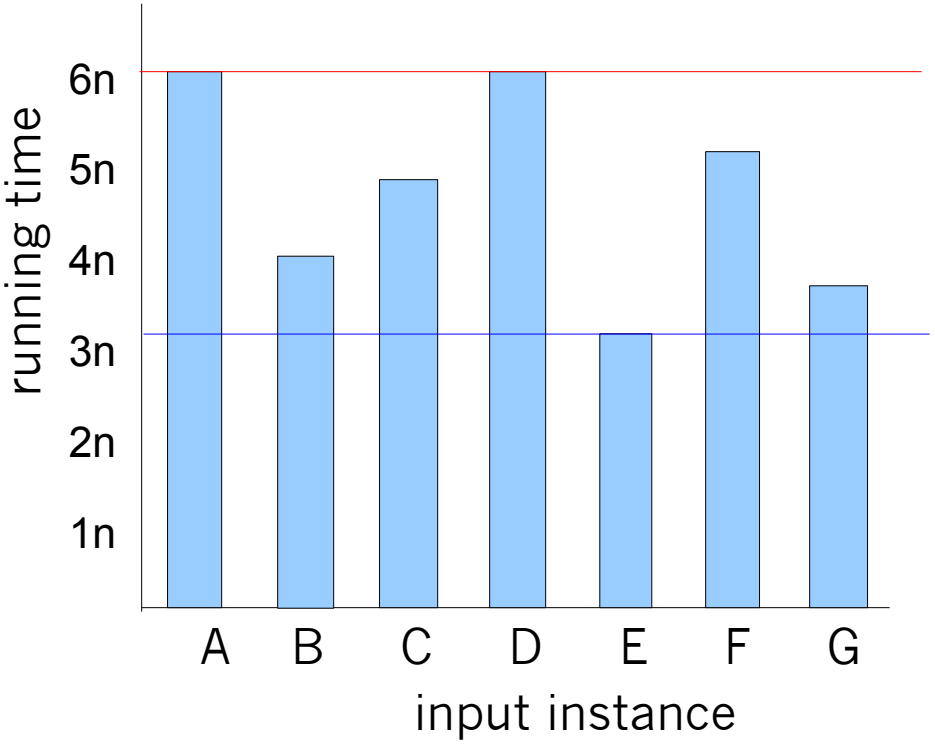
Worst/Average/Best Case

- 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: $t_j=j/2$, total running time is *quadratic* (time = an^2+bn+c)
  - Best case: elements already sorted, $t_j=1$, innermost loop is zero, 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:



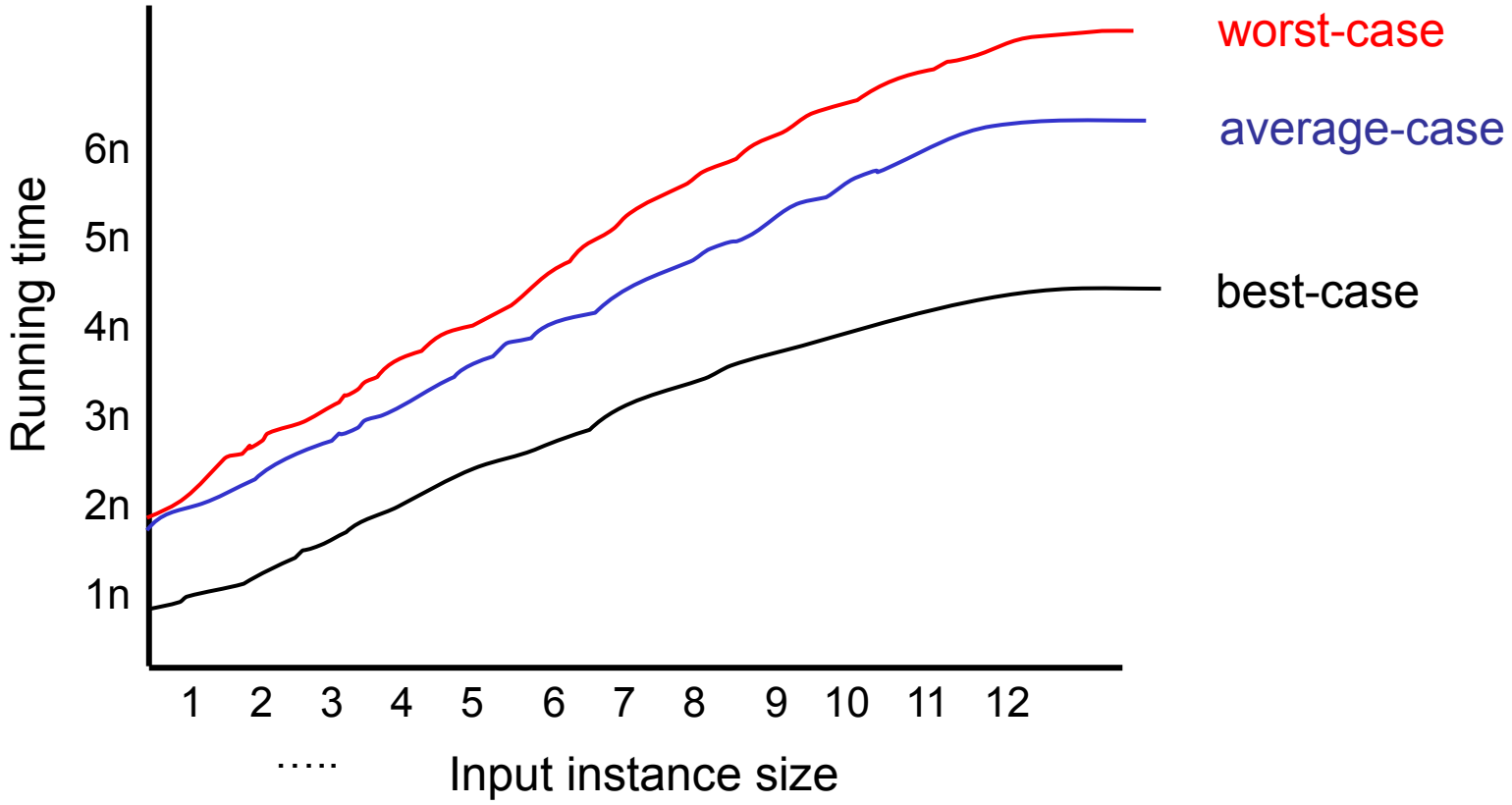
worst case = 4 3 2 1

average case ???

best case = 1 2 3 4

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] – a sorted array of integers,
           q – an integer.
OUTPUT: j s.t. A[j]=q. NIL if ∀j(1≤j≤n): A[j]≠q

j := 1
while j <= n and A[j] != q do j++
if j <= n then return j
else return NIL
```

- Worst case running time: *n*
- Average case running time: *n/2* (if *q* is present)

… under which assumption?

Binary Search

- Idea: Left and right bounds l, r.
  Elements to the right of r are bigger than search element, ...
- In each step, the range of the search space is cut in half

***INPUT*: A[1..n] – sorted (increasing) array of integers, *q* – integer.**
OUTPUT: an index *j* such that A[*j*] = *q*. *NIL*, if $\forall j\ (1 \le j \le n)$: A[*j*] $\neq q$

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

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 = ??
  - *n* = 2M, log 2M = ??
  - *n* = 20M, log 20M = ??

Suggested Exercises

- Implement binary search in 3 versions:
  - as in previous slides
  - without return statements inside the loop
  - Recursive
- As before, returning nil if q<a[l] or q>a[r]
  (trace the different executions)
- Implement a function printSubArray printing only the subarray from l to r, leaving blanks for the others
  - use it to trace the behaviour of binary search

DSA, Part 2: Overview

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

Asymptotic Analysis

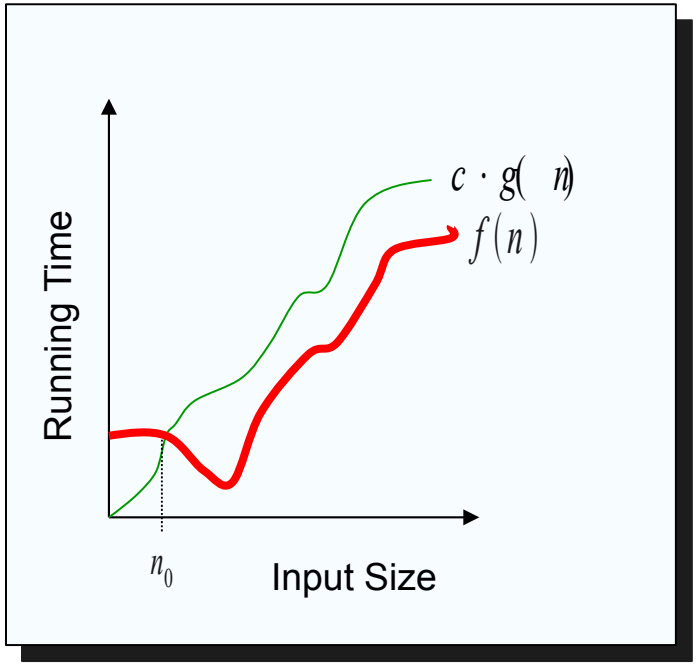
- Goal: simplify the analysis of the running time by getting rid of details, which are affected by specific implementation and hardware
  - "rounding" of numbers: 1,000,001 $\approx$ 1,000,000
  - "rounding" of functions: $3n^2 \approx n^2$
- Capturing the essence: how the running time of an algorithm 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

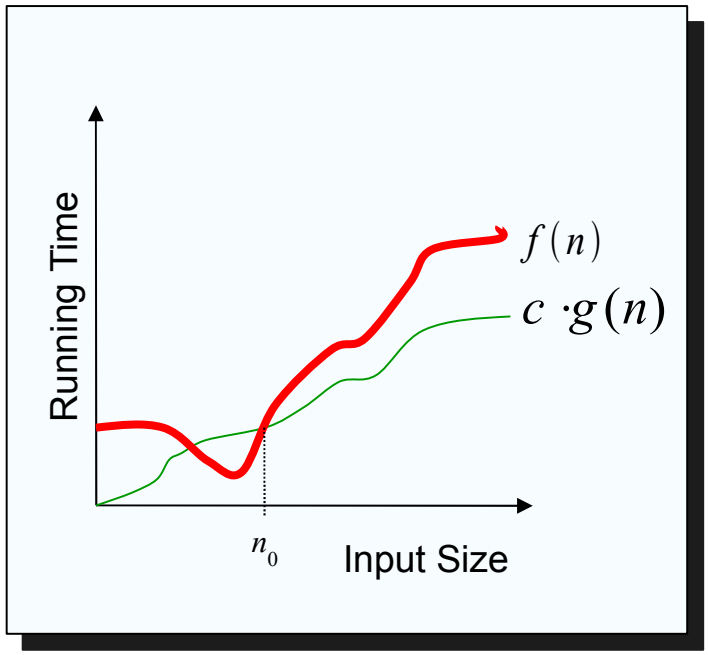
- Simple Rule: We can always drop lower order terms and constant factors, without changing big Oh:
  - 50 n log n is $O(n \log n)$
  - $7n - 3$ is $O(n)$
  - $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

- Note:
  - 50 n log n is $O(n^2)$
  - 50 n log n is $O(n^{100})$

  but this is less informative than saying
  - 50 n log n is $O(n \log n)$

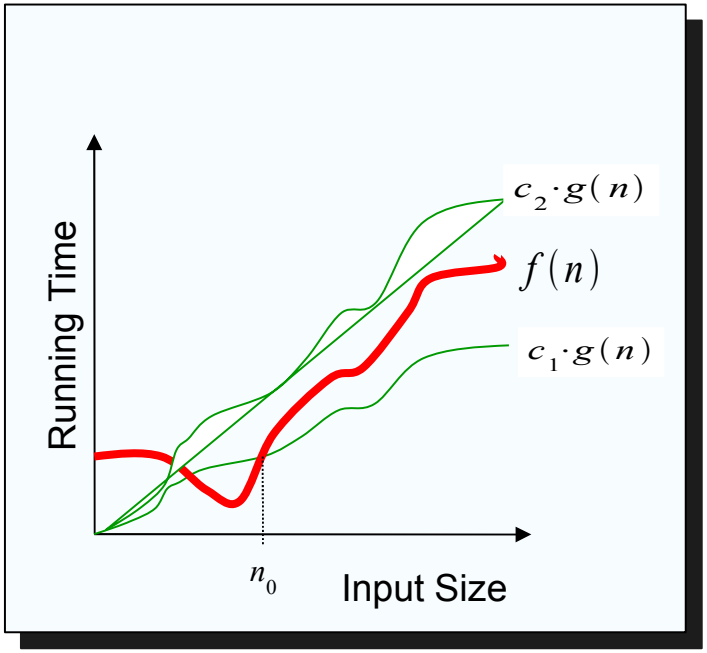
Asymptotic Notation/3

- 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 an unsorted array with search2 is $\Omega(n)$, with search1 it is $\Omega(\log n)$



Asymptotic Notation/4

- The "big-Theta" Θ-Notation
  - asymptoticly 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*

Two More Asymptotic Notations

f(n) is *much* smaller than *g(n)*

- "Little-Oh" notation $f(n)=o(g(n))$
  non-tight analogue of Big-Oh
  - For every $c > 0$, there exists $n_0>0$, s.t.

    $$f(n) < c\, g(n)$$

    for $n \geq n_0$
  - If $f(n) = o(g(n))$, it is said that *g(n) dominates f(n)*

f(n) is *much* bigger than *g(n)*

- "Little-omega" notation $f(n)=\omega(g(n))$
  non-tight analogue of Big-Omega

Asymptotic Notation/6

- Analogy with real numbers
  - $f(n) = O(g(n)) \cong f \leq g$
  - $f(n) = \Omega(g(n)) \cong f \geq g$
  - $f(n) = \Theta(g(n)) \cong f = g$
  - $f(n) = o(g(n)) \cong f < g$
  - $f(n) = \omega(g(n)) \cong f > g$
- Abuse of notation:

  $f(n) = O(g(n))$ actually means

  $f(n) \in O(g(n))$

Comparison of Running Times

Determining the maximal problem size

Running Time T(n) in μs	1 second	1 minute	1 hour
$400n$	2500	150'000	9'000'000
$20n \log n$	4096	166'666	7'826'087
$2n^2$	707	5477	42'426
n^4	31	88	244
2^n	19	25	31

DSA, Part 2: Overview

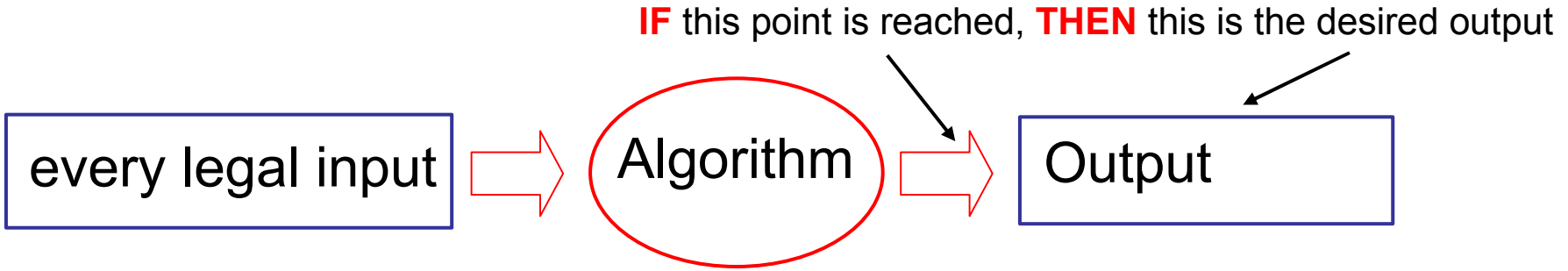
- Complexity of algorithms
- Asymptotic analysis
- Correctness of algorithms
- Special case analysis

Correctness of Algorithms

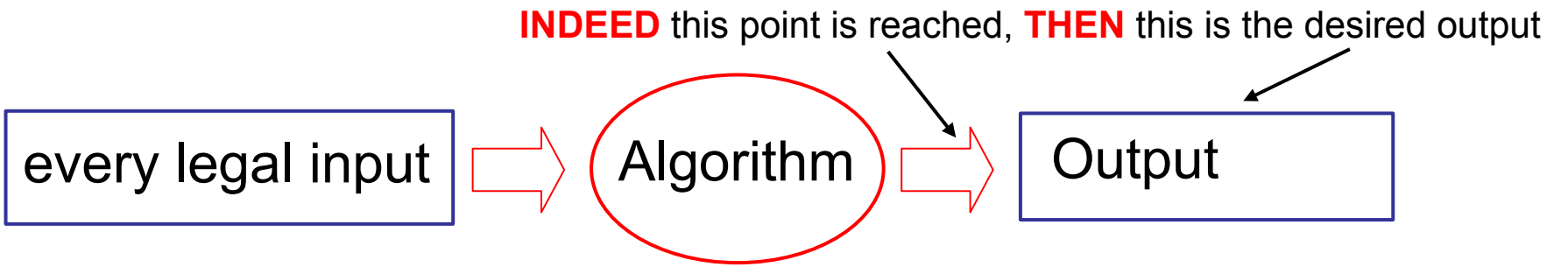
- An algorithm is *correct* if for every legal input, it terminates and produces the desired output.
- Automatic proof of correctness is not possible.
- There are practical techniques and rigorous formalisms that help to reason about the correctness of (parts of) algorithms.

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], …, *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)

Loop Invariants

- **Invariants:** assertions that are valid every time they are reached (many times during the execution of an algorithm, e.g., in loops)
- 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 NIL 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; r := n;
do
   m := ⌊(l+r)/2⌋
   if A[m]=q then return m
   else if A[m]>q then r := m-1
   else l := m+1
while l <= r
return NIL
```

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

Example: Binary Search/2

- **Invariant:**
  ∀i∈[1..l-1]: A[i]<q (ia)
  ∀i∈[r+1..n]: A[i]>q (ib)

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

- **Maintenance**: 1 ≤ l, r ≤ n, m = ⌊(l+r)/2⌋
  - A[m] != q & q < A[m] implies r = m-1
    A sorted implies ∀k∈[r+1..n]: A[k] > q (ib)
  - A[m] != q & A[m] < q implies l = m+1
    A sorted implies ∀k∈[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;
do
   m := ⌊(l+r)/2⌋
   if A[m]=q then return m
   else if A[m]>q then r := m-1
   else l := m+1
while l <= r
return NIL
```

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

  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 $l \le r$ no longer holds

Example: Insertion Sort/1

```
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
```

Loop invariants:

External "for" loop

A[1..j-1] is sorted

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

Internal "while" loop

- A[1...i], key, A[i+1…j-1]
- A[1..i] is sorted
- A[i+1..j-1] is sorted
- A[k] > key for all k in {i+1,...,j-1}

Example: Insertion Sort/2

External for loop:

(i) A[1...j-1] is sorted

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

Internal while loop:

- A[1...i], key, A[i+1…j-1]
- A[1..i] is sorted
- A[i+1..j-1] is sorted
- A[k] > key for all k in {i+1,...,j-1}

```
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:

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

i=j-1: A[1...j-1], key, A[j...j-1] where key=A[j]

A[j...j-1] is empty (and thus trivially sorted)

A[1...j-1] is sorted (invariant of outer loop)

Example: Insertion Sort/3

External for loop:

(i) A[1...j-1] is sorted

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

Internal while loop:

- A[1...i], key, A[i+1…j-1]
- A[1..i] is sorted
- A[i+1..j-1] is sorted
- A[k] > key for all k in {i+1,...,j-1}

```
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: A to A'

- (A[1...j-1] sorted) and (insert A[j]) implies (A[1...j] sorted)
- A[1...i-1], key, A[i,i+1...j-1] satisfies conditions because of condition A[i] > key and A[1...j-1] being sorted.

Example: Insertion Sort/4

External for loop:

(i) A[1...j-1] is sorted

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

Internal while loop:

- A[1...i], key, A[i+1…j-1]
- A[1..i] is sorted
- A[i+1..j-1] is sorted
- A[k] > key for all k in {i+1,...,j-1}

```
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:

- main loop, j=n+1: A[1...n] sorted.
- A[i]≤key: (A[1...i], key, A[i+1...j-1]) = A[1...j-1] is sorted
- i=0: (key, A[1...j-1]) = A[1...j] is sorted.

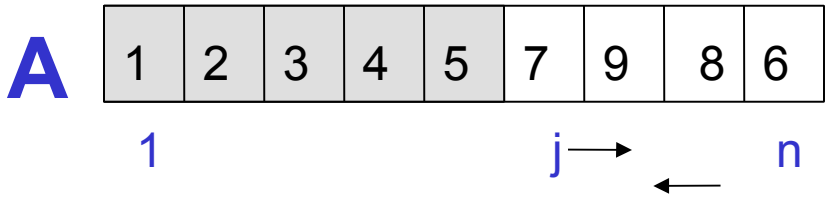
Example: Bubble Sort

```
INPUT: A[1..n] – an array of integers
OUTPUT: permutation of A s.t. A[1]≤A[2]≤… ≤A[n]

for j := 2 to n do
  for i := n downto j do
    if A[i-1] > A[i] then
      value := A[i-1];
      A[i-1] := A[i];
      A[i]:= value
```

- 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)

Example: Bubble Sort



Strategy

- Start from the back and compare pairs of adjacent elements.
- Swap the elements if the larger comes before the smaller.
- In each step the smallest element of the unsorted part is moved to the beginning of the unsorted part and the sorted part grows by one.

44 55 12 42 94 18 06 67
06 44 55 12 42 94 18 67
06 12 44 55 18 42 94 67
06 12 18 44 55 42 67 94
06 12 18 42 44 55 67 94
06 12 18 42 44 55 67 94
06 12 18 42 44 55 67 94
06 12 18 42 44 55 67 94

Loop Invariants for Bubble Sort

- Outer loop: "A[1..j-2] sorted and contains the j-2 smallest values of the array"

  Note: loop finishes with j = n+1

  In the end:

  A[1..n-1] sorted and minimum,

  hence, A[1..n] is sorted

- Inner loop: "A[i] is the minimum in A[i..n]"

  Note: loop finishes with i = j-1

  In the end:

  A[j-1] is the minimum in A[j-1..n], which implies the outer loop invariant

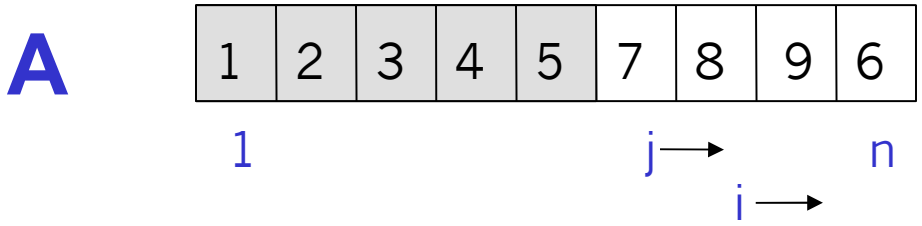
Example: Selection Sort

```
INPUT: A[1..n] – an array of integers
OUTPUT: a permutation of A such that A[1]≤ A[2]≤ …≤A[n]

for j := 1 to n-1 do
  min := A[j]; minpos := j
  for i := j+1 to n do
    if A[i] < min then min := A[i]; minpos := i;
  A[minpos] := A[j]; A[j] := min
```

- What is a good loop invariant for the outer loop?
- … and what is a good loop invariant for the inner loop?

Example: Selection Sort



Strategy

- Start empty handed.
- Enlarge the sorted part by swapping the first element of the unsorted part with the smallest element of the unsorted part.
- Continue until the unsorted part consists of one element only.

44 55 12 42 94 18 06 67
06 55 12 42 94 18 44 67
06 12 55 42 94 18 44 67
06 12 18 42 94 55 44 67
06 12 18 42 94 55 44 67
06 12 18 42 44 55 94 67
06 12 18 42 44 55 94 67
06 12 18 42 44 55 67 94

Loop Invariants for Selection Sort

- Outer loop: "A[1..j-1] sorted and contains
  the j-1 smallest values of the array"

  Note: loop finishes with j = n

  In the end: A[1..n-1] sorted and minimum,
  hence, A[1..n] is sorted

- Inner loop: "min holds the minimum of A[j..i-1] and
  minpos holds the position of the minimum"

  Note: loop finishes with i = n+1

  In the end: min holds the minimum of A[j..n]
  then, swap(minpos,j) puts min into j,
  which implies the outer loop invariant

Exercise

- Apply the same approach to prove the correctness of bubble sort.

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

- zero, empty string
- negative number
- border of domain

- entering a function
- termination of a function

- 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] ≥ 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] ≥ A[i+1] then return FALSE
return TRUE
```

- Start of loop, i=1 ➔ OK
- End of loop, i=n ➔ 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] ≥ A[i+1] then return FALSE
return TRUE
```

- Start of loop, i=1 ➔ OK
- End of loop, i=n-1 ➔ OK
- A=[1,1,1] ➔ First iteration, from i=1 to i=2 ➔ OK
- A=[1,1,1] ➔ 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
  if A[i] > A[i+1] then return FALSE
return TRUE
```

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

Binary Search, Variant 1

Analyze the following algorithm by considering special cases.

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

Binary Search, Variant 1

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

- 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 := ⌊(l+r)/2⌋
   if A[m] = q then return m
   else if A[m] > q then r := m-1
   else l := m+1
while l <= r
return NIL
```

- Start of loop ➔ OK
- End of loop, l=r ➔ OK
- First iteration ➔ OK
- A=[1,1,1] ➔ OK
- Empty data structure, n=0 ➔ Error! Tries to access A[0]
- One-element data structure, n=1 ➔ OK

Binary Search, Variant 1

```
l := 1; r := n
If r < 1 then return NIL;
do
   m := ⌊(l+r)/2⌋
   if A[m] = q then return m
   else if A[m] > q then r := m-1
   else l := m+1
while l <= r
return NIL
```

- 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 := ⌊(l+r)/2⌋
  if A[m] <= q
    then l := m+1 else r := m
if A[l-1] = q
  then return q else return NIL
```

Binary Search, Variant 3

Analyze the following algorithm by considering special cases

```
l := 1; r := n
while l <= r do
   m := ⌊(l+r)/2⌋
   if A[m] <= q
      then l := m+1 else r := m
if A[l-1] = q
   then return q else return NIL
```

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] ≤ A[2] ≤ … ≤ 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..n1], B[1..n2] sorted arrays of
integers
OUTPUT: permutation C of A.B s.t.
  C[1] ≤ C[2] ≤...≤ C[n1+n2]

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..n1], B[1..n2] sorted arrays of
integers
OUTPUT: permutation C of A.B s.t.
  C[1] ≤ C[2] ≤...≤ C[n1+n2]

i:=1;j:=1;
for k:= 1 to n1 + n2 do
      if j > n2 or (i <= n1 and A[i]<=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] ≤ B[j]" decides
- B exhausted, A not, implies
  j > n2, implies A wins
- B not exhausted, A exhausted, implies
  j ≤ n2 && i > n, implies B wins
- A, B both exhausted, implies
  k > n1+n2, implies algorithm stops

Math Refresher

- Arithmetic progression

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

- Geometric progression (for a number a ≠ 1)

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

Induction Principle

We want to show that property P is true for all integers $n \geq n0$.

Basis: prove that P is true for $n0$.

Inductive step: prove that if P is true for all k such that $n0 \leq k \leq n - 1$ then P is also true for n.

Exercise: Prove that every Fibonacci number of the form 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