Data Structures and Algorithms

Roberto Sebastiani
roberto.sebastiani@disi.unitn.it
http://www.disi.unitn.it/~rseba

- Week 02 -

B.S. In Applied Computer Science
Free University of Bozen/Bolzano
academic year 2010-2011

Acknowledgements & Copyright Notice

Data Structures and Algorithms
Week 2

1. Complexity of algorithms
2. Asymptotic analysis
3. Correctness of algorithms
4. Special case analysis

Data Structures and Algorithms
Week 2

1. Complexity of algorithms
2. Asymptotic analysis
3. Correctness of algorithms
4. 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

- Time to compute the **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	$\sum_{j=2}^{n} t_j$
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

Analysis of Insertion Sort/2

- The running time of an algorithm 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

- Often 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

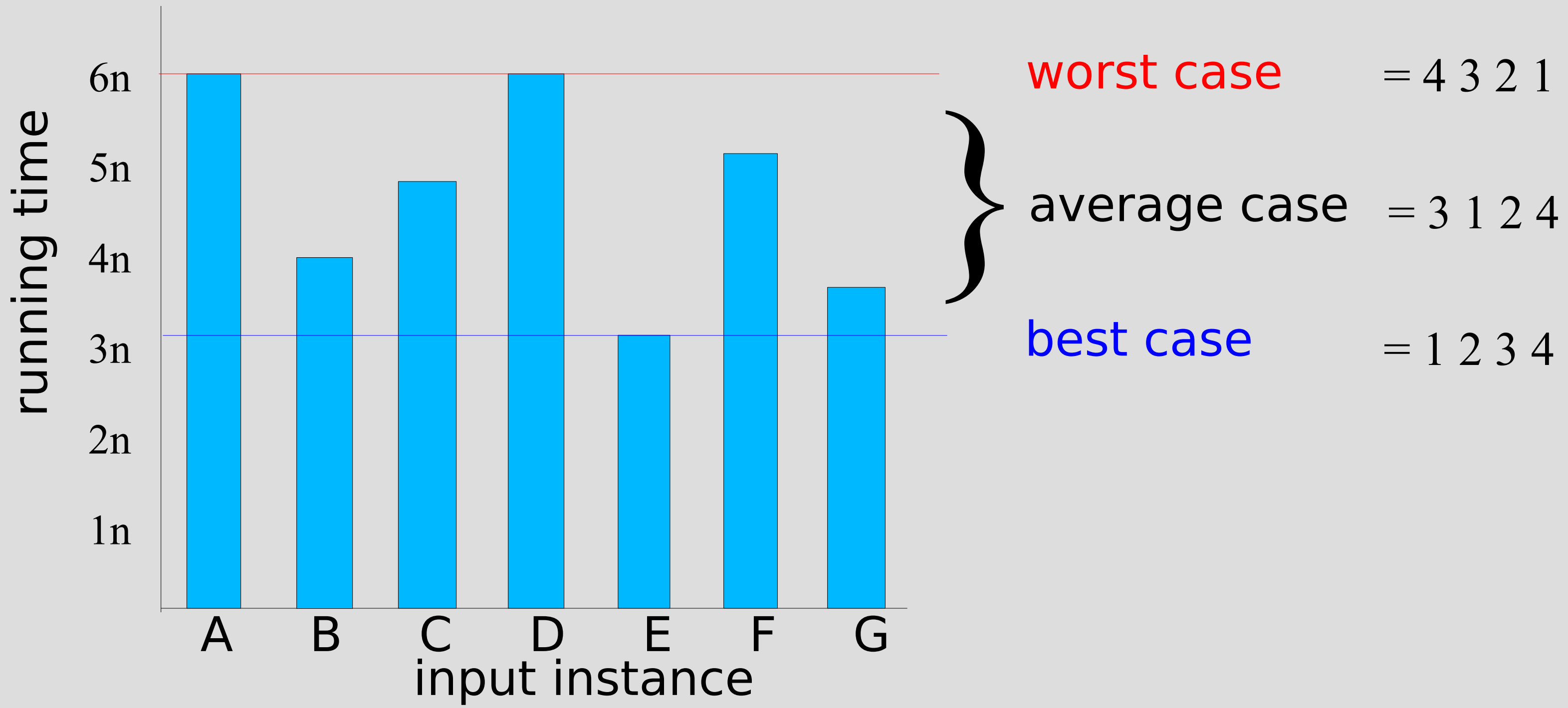
- Often it is sufficient to count the number of iterations of the core (innermost) part.
  - No distinction between comparisons, assignments, etc (that means roughly the same cost for all of them).
  - Gives precise enough results.
- In some cases the cost of selected operations dominates all other costs.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

Best/Worst/Average Case

- Analyzing insertion sort's $\sum_{j=2}^{n}(t_j - 1)$
  - **Best case**: elements already sorted, $t_j = 1$, innermost loop is zero, total running time is *linear* (time = an+b).
  - **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).

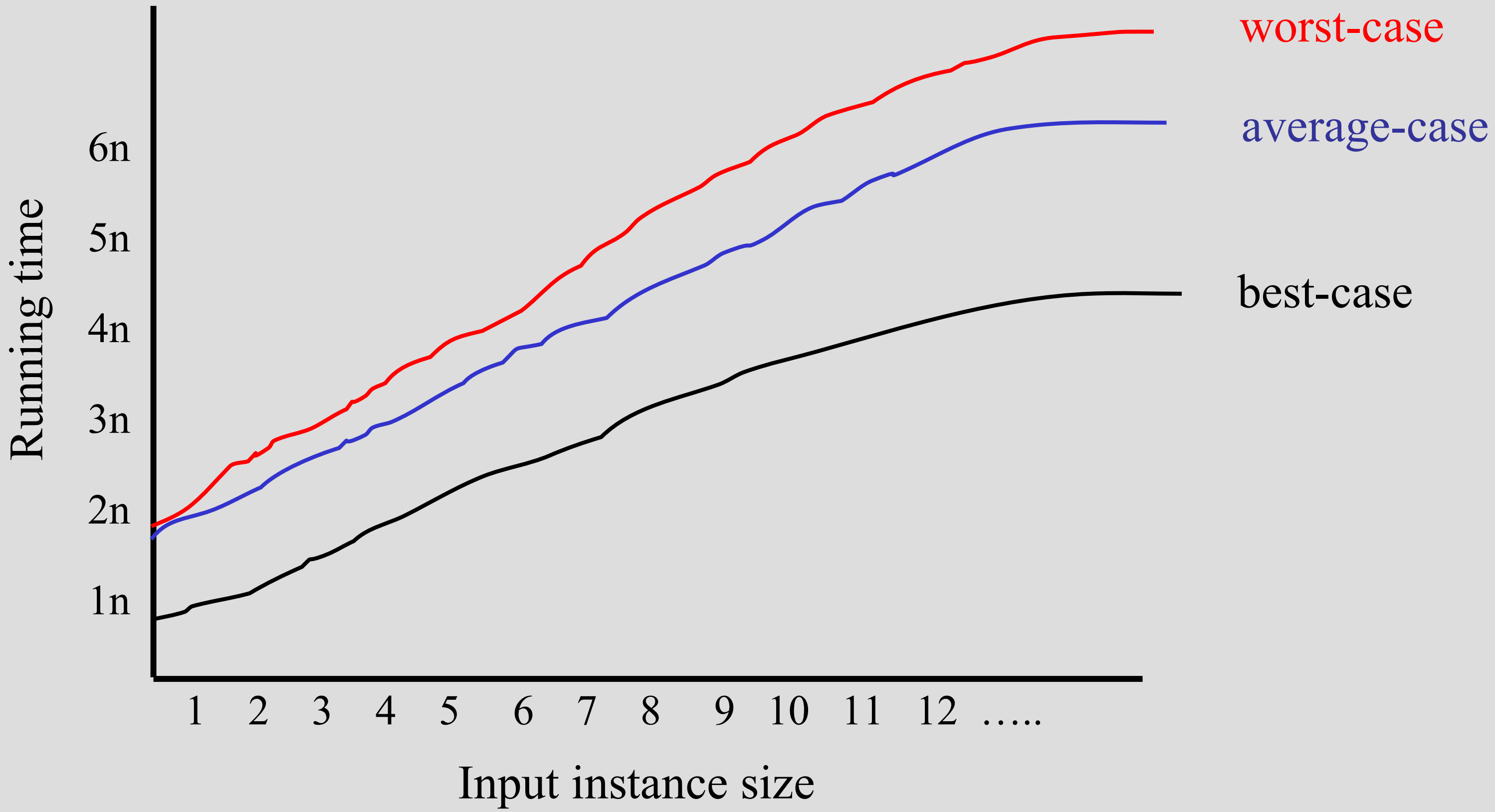
Best/Worst/Average Case/2

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



Best/Worst/Average Case/3

– For inputs of all sizes:



Best/Worst/Average Case/4

- **Worst case** is usually 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**.
  - Finding the **average case** can be very difficult.

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 present)
- Best case running time: 1

Binary Search

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

INPUT: A[1..n] – sorted (increasing) array of integers, *q* – integer.
OUTPUT: an index *j* such that A[*j*] = *q*. *NIL*, if ∀*j* (1≤*j*≤*n*): A[*j*] ≠ *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 the loop is 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:
  - $n = 200'000$, $\log n = \log 200'000 = 18$
  - $n = 2M$, $\log 2M = 21$
  - $n = 20M$, $\log 20M = 24$

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

Data Structures and Algorithms
Week 2

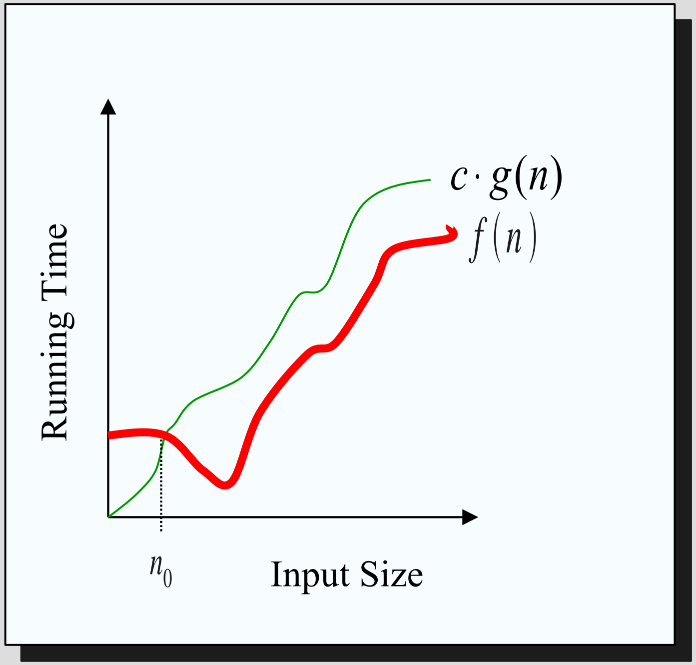
1. Complexity of algorithms
2. Asymptotic analysis
3. Correctness of algorithms
4. Special case analysis

Asymptotic Analysis

- Goal: to 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 ≈ 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
  - asymptotic upper bound
  - $f(n) = O(g(n))$ iff there exists constants $c>0$ and $n_0>0$, *s.t.* $\mathbf{f(n) \leq c\ g(n)}$ for $n \geq n_0$
  - $f(n)$ and $g(n)$ are functions over non-negative integers
- Used for *worst-case* analysis

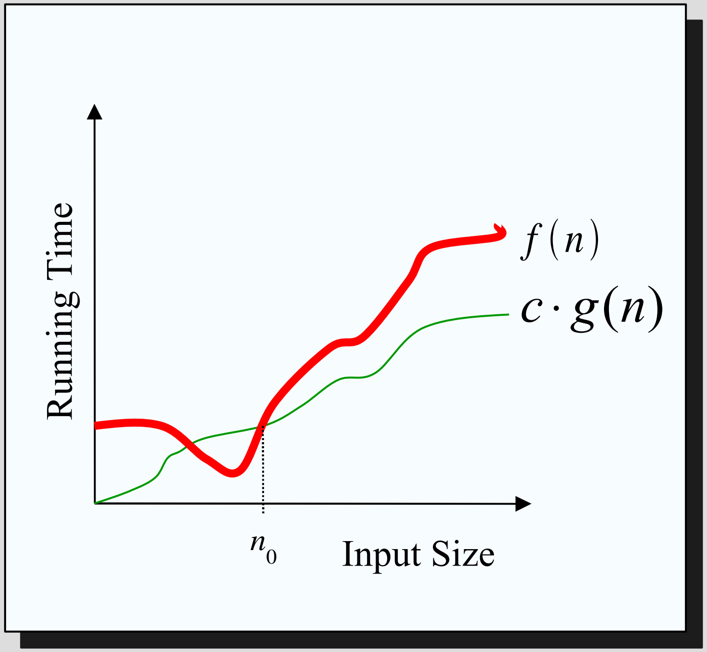


Asymptotic Notation/2

- Simple Rule: Drop lower order terms and constant factors.
  - $50\, n \log n$ is $O(n \log n)$
  - $7n - 3$ is $O(n)$
  - $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$
- Note: Although $(50\, n \log n)$ is also $O(n^2)$, or even $O(n^{100})$, it is expected that an approximation is of the smallest possible order.

Asymptotic Notation/3

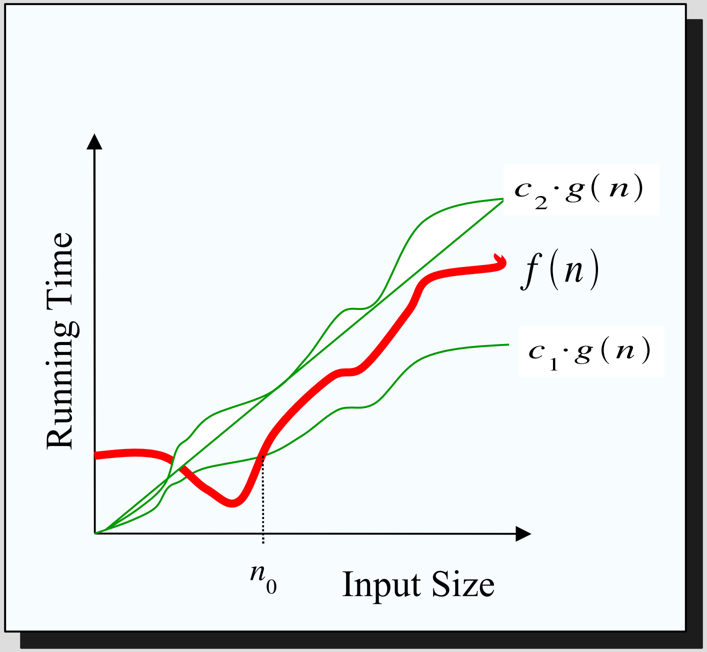
- The "big-Omega" Ω–Notation
  - asymptotic lower bound
  - $f(n) = \Omega(g(n))$ iff there exists constants $c>0$ and $n_0>0$, *s.t.* $\mathbf{c\ g(n) \leq f(n)}$ for $n \geq n_0$
- Used to describe *best-case* running times or lower bounds of algorithmic problems.
  - E.g., searching in an unsorted array with search2 is $\Omega(n)$, *with search1 it is* $\Omega(1)$



Asymptotic Notation/4

3

- The "big-Theta" Θ–Notation
  - asymptoticly tight bound
  - $f(n) = \Theta(g(n))$ if there exists constants $c_1>0$, $c_2>0$, and $n_0>0$, *s.t.* for $n \geq n_0$
    $\mathbf{c_1\, g(n) \leq f(n) \leq c_2\, g(n)}$



- $f(n) = \Theta(g(n))$ iff
  $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- *Note:* $O(f(n))$ is often abused instead of $\Theta(f(n))$

Asymptotic Notation/5

- Two more asymptotic notations
  - "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. $\mathbf{f(n) < c\, g(n)}$ for $n \geq n_0$
    - If $f(n)=o(g(n))$, it is said that $g(n)$ *dominates* $f(n)$.
  - "Little-omega" notation $f(n)=\omega(g(n))$ non-tight analogue of Big-Omega

Asymptotic Notation/6

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

Data Structures and Algorithms
Week 2

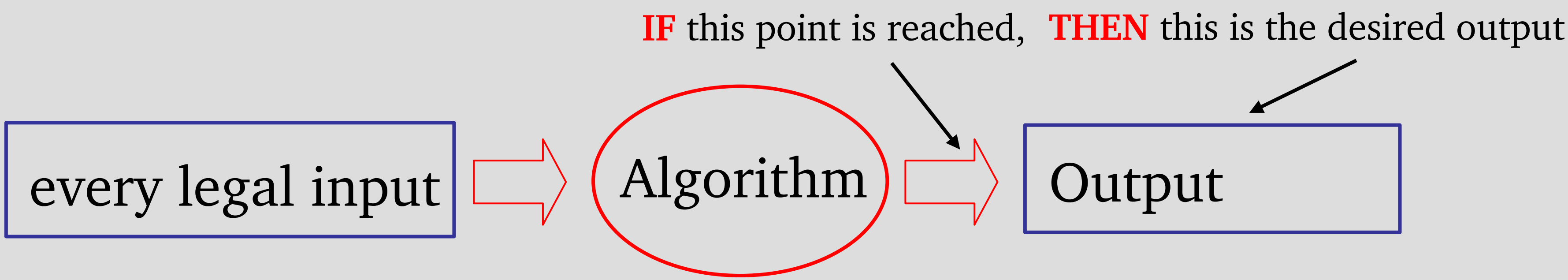
1. Complexity of algorithms
2. Asymptotic analysis
3. Correctness of algorithms
4. Special case analysis

Correctness of Algorithms

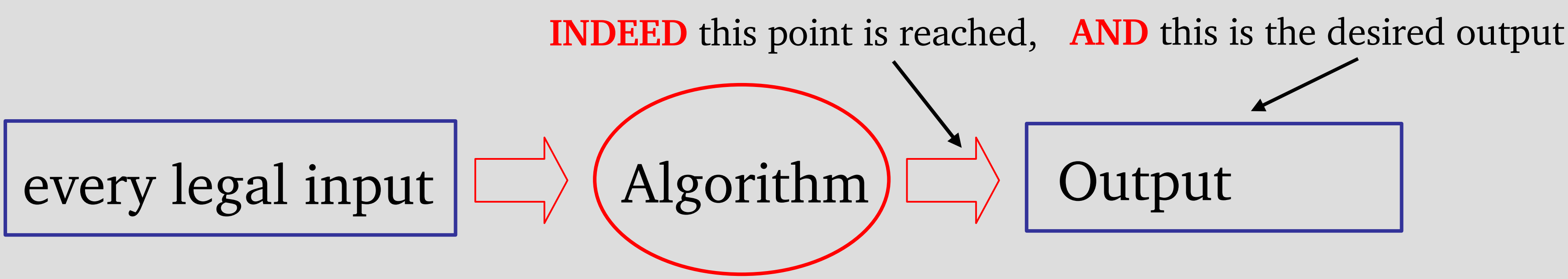
- An algorithm is *correct* if for every legal input, it terminates and produces the desired output.
- Automatic proof of correctness is not possible.
- There are practical techniques and rigorous formalisms that help to reason about the correctness of (parts of) algorithms.

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

```
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
```

- 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 yields $\forall j,i \in [1..0]$: A[i]<q
  this holds because [1..0] is empty
- r=n yields $\forall j,i \in [n+1..n]$: A[i]>q
  this holds because [n+1..n] is empty

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;
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**: l, r, $m = \lfloor (l+r)/2 \rfloor$
- A[m] != q & A[m] > q, r = m-1, A sorted implies
  $\forall k \in [r+1..n]: A[k] > q$ (ib)
- A[m] != q & A[m] < q, l = m+1, A sorted implies
  $\forall k \in [1..l-1]: A[k]<q$ (ia)

Example: Binary Search/3

- **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
```

- **Termination**: l, r, l<=r
- Two cases:
  - l := m+1 we get $\lfloor(l+r)/2\rfloor+1 > l$
  - r := m-1 we get $\lfloor(l+r)/2\rfloor-1 < r$
- The range gets smaller during each iteration and the loop will terminate when l<=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
  i. A[1...j-1] is sorted
  ii. A[1...j-1] $\in$ A^{orig}
- Internal "while" loop:

A[1..j]: A[1...i]A[i+1]A[i+2…j], where A[i+1] is a placeholder for key, s.t.:

a) A[i+2…j] is sorted

b) A[1...i] is sorted

c) key <= A[k] forall k in {i+2...j},

d) A[i] <= A[k] forall k in {i+2...j}

Example: Insertion Sort/2

- External for loop
  - i. A[1...j-1] is sorted
  - ii.A[1...j-1] ∈ A^{orig}
- Internal while loop:
  A[1...i]A[i+1]A[i+2…j] s.t.
  a)A[i+2…j] is sorted
  b)A[1...i] is sorted
  c)key<=A[k] forall k in {i+2...j},
  d)A[i]<=A[k] forall k in {i+2...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:

- j=2: A[1…1] ∈ A^{orig} and is trivially sorted
- i=j-1: A[1...i], key, A[i+2,...j] s.t. key=A[j]
  a)A[i+2…j] is empty, and thus trivially sorted,
  b)A[1...i] is sorted (invariant of outer loop)
  c)trivial since {i+2...j} empty
  d)trivial since {i+2...j} empty

Example: Insertion Sort/3

- External for loop
  i. A[1...j-1] is sorted
  ii. A[1...j-1] $\in$ A^{orig}
- Internal while loop:
  A[1…i]A[i+1]A[i+2…j] s.t.
  a)A[i+2…j] is sorted
  b)A[1...i] is sorted
  c)key<=A[k] forall k in {i+2...j},
  d)A[i]<=A[k] forall k in {i+2...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: A $\longrightarrow$ A'

- If i.,ii. then A'[1…j]$\in$$A^{orig}$ sorted (by termination of internal while loop)
- i = i-1, A'[1…i-1]=A[1…i-1], and A'[i+1...j]=A[i]A[i+2...j]
  a)If a) then A'[i+1...j] sorted because of d)
  b)If b) then obviously A'[1…i-1] sorted
  c)If c) then key<=A'[k] forall k in {i+1,...j}
  d)If d) then A[i-1] <=A'[k] forall k in {i+1,...j} because of a)

Example: Insertion Sort/4

- External for loop
  i. A[1...j-1] is sorted
  ii. A[1...j-1] ∊ A^{orig}
- Internal while loop:
  A[1...i]A[i+1]A[i+2…j] s.t.
  a)A[i+2…j] is sorted
  b)A[1...i] is sorted
  c)key<=A[k] forall k in {i+2...j},
  d)A[i]<=A[k] forall k in {i+2...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:

- j=n+1: due to i. and ii. A[1...n] is sorted and A[1...n] ∊ A^{orig}
- A[1...i]A[i+1]A[i+2…j] s.t. i<=0 or A[i]<=key, a)-d) hold
  thus, after “A[i+1]:=key” A[1..j] is sorted

Suggested exercises

- Apply the same process to prove the correctness of insertion, selection and bubble sort.
- Do the same also for the versions of the algorithms in reverse order.
- Add to the implementations of the above algorithms, for both inner and outer loops, a call to some method which aborts if the loop invariant is violated

Data Structures and Algorithms
Week 2

1. Complexity of algorithms
2. Asymptotic analysis
3. Correctness of algorithms
4. Special case analysis

Special Case Analysis

- Consider extreme cases and make sure our 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 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 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 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
- 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 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
- 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, Variant1

- 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, Variant1

```
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! Ex: search 3 in [3 5 7]

Binary Search, Variant1

```
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, Variant1

```
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

Suggested exercises

- Apply the same special-case analysis to the two versions of binary search in next slides
- Define an algorithm to merge two sorted arrays into one:
  - Describe its complexity
  - Prove its correctness via loop invariants
  - Analyze special cases (be careful!)

Binary Search, Variant2

- 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, Variant3

- 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
```

Insert 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=i+1;
      Else C[k]=B[j];j=j+1;
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=i+1;
      Else C[k]=B[j];j=j+1;
Return C;
```

Summary

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

Next Week

- Divide-and-conquer
- Merge sort
- Writing recurrences to analyze the running time of recursive algorithms.