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
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).

```
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
```

<table>
<thead>
<tr>
<th>Cost</th>
<th>Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>n</td>
</tr>
<tr>
<td>c2</td>
<td>n-1</td>
</tr>
<tr>
<td>c3</td>
<td>n-1</td>
</tr>
<tr>
<td>c4</td>
<td>(\sum_{j=2}^{t_j} t_j)</td>
</tr>
<tr>
<td>c5</td>
<td>(\sum_{j=2}^{\infty} (t_j-1))</td>
</tr>
<tr>
<td>c6</td>
<td>(\sum_{j=2}^{\infty} (t_j-1))</td>
</tr>
<tr>
<td>c7</td>
<td>n-1</td>
</tr>
</tbody>
</table>

\(t_j\) 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

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{j=2}^{n} t_j$$
$$+ c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 (n - 1)$$
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**: $= 1 \ 2 \ 3 \ 4$

![Diagram of running times for different input instances A to G, with worst case values indicated.]
Worst/Average/Best Case/3

For inputs of all sizes:

![Graph showing running time vs. input instance size with lines for worst-case, average-case, and best-case complexities.]

Worst-case
Average-case
Best-case
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


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

\[
\begin{align*}
j & := 1 \\
\text{while } j \leq n \text{ and } A[j] \neq q \text{ do } j++ \\
\text{if } j \leq n \text{ then return } j \\
\text{else return } NIL
\end{align*}
\]

- 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 $\forall j (1 \leq j \leq n): A[j] \neq q$

```
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 $l \leq r$
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. $f(n) \leq c \cdot g(n)$ 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” $\Omega$-Notation
  – asymptotic lower bound
  – $f(n) = \Omega(g(n))$ iff there exist $c > 0$ and $n_0 > 0$, s.t. $c \cdot g(n) \leq f(n)$, 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” \( \Theta \)-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
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.
    $$f(n) < c \cdot g(n)$$
    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

\( f(n) \) is *much* smaller than \( g(n) \)
\( f(n) \) is *much* bigger than \( g(n) \)
Asymptotic Notation/6

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

- 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

<table>
<thead>
<tr>
<th>Running Time T(n) in $\mu$s</th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>$400n$</td>
<td>2500</td>
<td>150'000</td>
<td>9’000’000</td>
</tr>
<tr>
<td>$20n \log n$</td>
<td>4096</td>
<td>166’666</td>
<td>7’826’087</td>
</tr>
<tr>
<td>$2n^2$</td>
<td>707</td>
<td>5477</td>
<td>42’426</td>
</tr>
<tr>
<td>$n^4$</td>
<td>31</td>
<td>88</td>
<td>244</td>
</tr>
<tr>
<td>$2^n$</td>
<td>19</td>
<td>25</td>
<td>31</td>
</tr>
</tbody>
</table>
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**
  - Every legal input
    - IF this point is reached, THEN this is the desired output

- **Total correctness**
  - Every legal input
    - INDEED this point is reached, THEN this is the desired output
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., $A[1], \ldots, A[j]$ 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:**
  \[ \forall i \in [1..l-1]: A[i] < q \quad (ia) \]
  \[ \forall i \in [r+1..n]: A[i] > q \quad (ib) \]

• **Initialization:** \( l = 1, r = n \)
  the invariant holds because there are no elements to the left of \( l \) or to the right of \( r \).
  \[ l = 1 \text{ yields } \forall i \in [1..0]: A[i] < q \]
  this holds because \([1..0]\) is empty
  \[ r = n \text{ yields } \forall i \in [n+1..n]: A[i] > q \]
  this holds because \([n+1..n]\) is empty

```
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 NIL
```
Example: Binary Search/2

• **Invariant:**
  \[ \forall i \in [1..l-1]: A[i] < q \quad (ia) \]
  \[ \forall i \in [r+1..n]: A[i] > q \quad (ib) \]

• **Maintenance:** \[ l \leq l, r \leq n, m = \lfloor (l+r)/2 \rfloor \]
  - \( A[m] \neq q \) \& \( q < A[m] \) implies \( r = m-1 \)
  \quad A \text{ sorted } \Rightarrow \forall k \in [r+1..n]: A[k] > q \quad (ib)
  - \( A[m] \neq q \) \& \( A[m] < q \) implies \( l = m+1 \)
  \quad A \text{ sorted } \Rightarrow \forall k \in [1..l-1]: A[k] < q \quad (ia)

```plaintext
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 NIL
```
Example: Binary Search/3

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

- **Termination:** \( 1 \leq l, r \leq n, l \leq r \)
  Two cases:
  
  \[
  l := m+1 \quad \text{implies} \quad l_{\text{new}} = \lfloor (l+r)/2 \rfloor + 1 > l_{\text{old}}
  \]
  
  \[
  r := m-1 \quad \text{implies} \quad r_{\text{new}} = \lfloor (l+r)/2 \rfloor - 1 < r_{\text{old}}
  \]

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

```plaintext
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 NIL
```
Example: Insertion Sort/1

Loop invariants:

External “for” loop
A[1..j-1] is sorted
A[1..j-1] ∈ 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[1..i] is sorted
– A[k] > key for all k in {i+1,...,j-1}

\[
\text{for } j := 2 \text{ to } n \text{ do}
\text{key := A[j]}
\text{i := j-1}
\text{while } i>0 \text{ and } A[i]>key \text{ do}
\text{A[i+1] := A[i]}
\text{i--}
\text{A[i+1] := key}
\]
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:
- \( A[1...i], \text{key}, A[i+1...j-1] \)
- \( A[1..i] \) is sorted
- \( A[i+1..j-1] \) is sorted
- \( A[1..i] \) is sorted
- \( A[k] > \text{key} \) for all \( k \in \{i+1,...,j-1\} \)

Initialization:
(i), (ii) \( j = 2: A[1..1] \in A^{\text{orig}}[1..1] \) and is trivially sorted
\( i = j-1: A[1...j-1], \text{key}, A[j...j-1] \) where \( \text{key}=A[j] \)
\( A[j...j-1] \) is empty (and thus trivially sorted)
\( A[1...j-1] \) is sorted (invariant of outer loop)

\[
\begin{array}{l}
\text{for } j := 2 \text{ to } n \text{ do} \\
\text{key := A}[j] \\
i := j-1 \\
\text{while } i>0 \text{ and } A[i]>\text{key} \text{ do} \\
i-- \\
A[i+1] := \text{key}
\end{array}
\]
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:
- $A[1...i], \text{key, } A[i+1...j-1]$
- $A[1..i]$ is sorted
- $A[i+1..j-1]$ is sorted
- $A[1..i]$ is sorted
- $A[k] > \text{key}$ for all $k$ in `{i+1,...,j-1}`

Maintenance: $A$ to $A'$
- ($A[1...j-1]$ sorted) and (insert $A[j]$) implies ($A[1...j]$ sorted)
- $A[1...i-1], \text{key, } A[i,i+1...j-1]$ satisfies conditions because of condition $A[i] > \text{key}$ and $A[1...j-1]$ being sorted.

```plaintext
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/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:
- $A[1..i]$, key, $A[i+1...j-1]$
- $A[1..i]$ is sorted
- $A[i+1..j-1]$ is sorted
- $A[1..i]$ is sorted
- $A[k] > \text{key}$ for all $k$ in $\{i+1,...,j-1\}$

**Termination:**
- main loop, $j=n+1$: $A[1...n]$ sorted.
- $A[i] \leq \text{key}$: $(A[1...i]$, key, $A[i+1...j-1]) = A[1...j-1]$ is sorted
- $i=0$: $(\text{key}, A[1...j-1]) = A[1...j]$ is sorted.

```plaintext
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
```
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
- 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.

\[
\begin{align*}
\text{INPUT: } & A[1..n] \text{ – an array of integers.} \\
\text{OUTPUT: } & \text{TRUE if } A \text{ is sorted; FALSE otherwise}
\end{align*}
\]

\[
\text{for } i := 1 \text{ to } n \\
\text{ if } A[i] \geq A[i+1] \text{ then return FALSE} \\
\text{return TRUE}
\]

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

```plaintext
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

```plaintext
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
Sortedness/3

**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
- First iteration, from i=1 to i=2 ⇒ OK
Sortededness/4

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

**OUTPUT**: TRUE if A is sorted; FALSE otherwise

```plaintext
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
- 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.

```plaintext
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

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

• Start of loop $\Rightarrow$ OK
• End of loop, $l=r \Rightarrow$ Error! Example: search 3 in [3 5 7]
Binary Search, Variant 1

\begin{verbatim}
l := 1; r := n
do
  \text{m := } \lfloor (l+r)/2 \rfloor
  \text{if } A[m] = q \text{ then return } m
  \text{else if } A[m] > q \text{ then } r := m-1
  \text{else } l := m+1
while \ l \leq r
return \ NIL
\end{verbatim}

• Start of loop \( \Rightarrow \) OK
• End of loop, \( l=r \Rightarrow \) OK
• First iteration \( \Rightarrow \) OK
• \( A=[1,1,1] \Rightarrow \) OK
• Empty data structure, \( n=0 \Rightarrow \) Error! Tries to access \( A[0] \)
• One-element data structure, \( n=1 \Rightarrow \) OK
Binary Search, Variant 1

```plaintext
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

```plaintext
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

\[
\text{l := 1; r := n}
\]

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

\[
\text{return } q
\]

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.  

```plaintext
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;
Math Refresher

- Arithmetic progression
  \[ \sum_{i=0}^{n} i = 0 + 1 + \ldots + n = n(n + 1)/2 \]

- Geometric progression (for a number \( a \neq 1 \))
  \[ \sum_{i=0}^{n} a^i = 1 + a^2 + \ldots + a^n = \frac{1 - a^{n+1}}{1 - a} \]
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