SCC0602 - Algoritmos e Estruturas de Dados I

Correctness

Professor: André C. P. L. F. de Carvalho, ICMC-USP
PAE: Rafael Martins D'Addio
Monitor:

Today

- Correctness of algorithms
- Growth of functions and asymptotic notation
- Revision of some basic math
- Conclusion

© André de Carvalho - ICMC/USP 2

Exercise

- Write a pseudocode of an algorithm to find the two smallest numbers in a sequence of numbers given as an array

© André de Carvalho - ICMC/USP 3

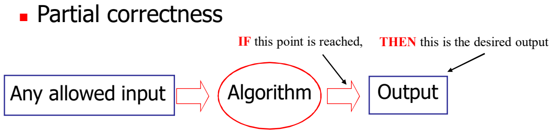
Correctness of Algorithms

- An algorithm is *correct* if
  - For any allowed input, it terminates and produces the desired output
- Automatic proof of correctness is not possible
  - But there are practical techniques and rigorous formalisms that help to reason about the correctness of algorithms

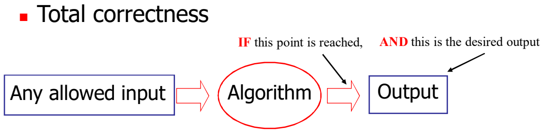
© André de Carvalho - ICMC/USP 4

Partial and Total Correctness

- Partial correctness



- Total correctness



© André de Carvalho - ICMC/USP 5

Correctness

- Difficult to prove
  - How to test for all possible inputs?
- Test algorithm with sample of possible inputs
  - Software testing
- Even more difficult is to prove total correctness

© André de Carvalho - ICMC/USP 6

Assertions

- To prove partial correctness
  - Associate a number of **assertions** (statements about the state of the execution) with specific checkpoints in the algorithm
  - E.g.: i=k, *A*[1], …,*A*[k] form an increasing sequence (IS)
- Other important assertions:
  - **Preconditions**
    - Assertions that **must be true *before*** the execution of an algorithm or a subroutine (INPUT)
  - **Postconditions**
    - Assertions that **must be true *after*** the execution of an algorithm or a subroutine (OUTPUT)

 7

Exercise

- Write a pseudocode of an algorithm to find the two smallest numbers in a sequence of numbers given as an array
  - Precondition:

 8

Exercise

- Write a pseudocode of an algorithm to find the two smallest numbers in a sequence of numbers given as an array
  - Precondition:
    - INPUT: an array of integers $A[1..n]$, $n > 0$

 9

Exercise

- Write a pseudocode of an algorithm to find the two smallest numbers in a sequence of numbers given as an array
  - Precondition:
    - INPUT: an array of integers $A[1..n]$, $n > 0$
  - Postcondition:

 10

Exercise

- Write a pseudocode of an algorithm to find the two smallest numbers in a sequence of numbers given as an array
  - Precondition:
    - INPUT: an array of integers $A[1..n]$, $n > 0$
  - Postcondition:
    - OUTPUT: (m_1, m_2), s. t. (such that) $m_1 < m_2$ and
      - For each $i \in [1..n]$, $m_1 \leq A[i]$ and, if $A[i] \neq m_1$, then $m_2 \leq A[i]$
      - If there is no m_2 satisfying these conditions, then …

 11

Exercise

- Write a pseudocode of an algorithm to find the two smallest numbers in a sequence of numbers given as an array
  - Precondition:
    - INPUT: an array of integers $A[1..n]$, $n > 0$
  - Postcondition:
    - OUTPUT: (m_1, m_2), s. t. (such that) $m_1 < m_2$ and
      - For each $i \in [1..n]$, $m_1 \leq A[i]$ and, if $A[i] \neq m_1$, then $m_2 \leq A[i]$
      - If there is no m_2 satisfying these conditions, then $m_2 = m_1$

 12

Loop Invariants

- Used to evaluate partial correctness
- Invariants: assertions (statements) that are valid any time they are reached
  - Are valid many times during the execution of an algorithm
    - E.g., in loops, a property or condition is true before and after each iteration

```
int a = 5;
int b = 0;
for (a > 0){
   a--;
   b++;
}
```

What is loop invariant in this algorithm?

© André de Carvalho - ICMC/USP 13

Loop Invariants

- Used to evaluate partial correctness
- Invariants: assertions (statements) that are valid any time they are reached
  - Are valid many times during the execution of an algorithm
    - E.g., in loops, a property or condition is true before and after each iteration

```
int a = 5;
int b = 0;
for (a > 0){
   a--;
   b++;
}
```

What is loop invariant in this algorithm?

a + b = 5

© André de Carvalho - ICMC/USP 14

Loop Invariants

- Three facts about a loop invariant:
  - **Initialization**
    - It is true before the first loop iteration
  - **Maintenance**
    - *If* it is true before a loop iteration, *then* it remains true before the next iteration
  - **Termination**
    - When the loop finishes, the invariant gives a useful property to show the correctness of the algorithm

© André de Carvalho - ICMC/USP 15

Binary Search

- Very simple search algorithm

Search for 47

0	4	7	10	14	23	45	47	53

https://brilliant.org/wiki/binary-search/

© André de Carvalho - ICMC/USP 16

Example: Binary Search (1)

- We want to make sure that if NIL is return *q* is not in A
- **Invariant**: *at the start of each **while** loop, q > A[i] for all i ∈[1..left-1] and q < A[i] for all i ∈[right+1..n]*

```
left←1
right←n
do
   j←⌊(left+right)/2⌋
   if A[j]=q then return j
   else if A[j]>q then right←j-1
   else left=j+1
while left<=right
return NIL
```

© André de Carvalho - ICMC/USP 17

Example: Binary Search (1)

- We want to make sure that if NIL is return *q* is not in A
- **Invariant**: *at the start of each **while** loop, q > A[i] for all i ∈[1..left-1] and q < A[i] for all i ∈[right+1..n]*

```
left←1
right←n
do
   j←⌊(left+right)/2⌋
   if A[j]=q then return j
   else if A[j]>q then right←j-1
   else left=j+1
while left<=right
return NIL
```

Floor

© André de Carvalho - ICMC/USP 18

Example: Binary Search (1)

- We want to make sure that if NIL is return *q* is not in A
- **Invariant**: *at the start of each **while** loop, q > A[i] for all i ∈[1..left-1] and q < A[i] for all i ∈[right+1..n]*

```
left←1
right←n
do
   j←⌊(left+right)/2⌋
   if A[j]=q then return j
   else if A[j]>q then right←j-1
   else left=j+1
while left<=right
return NIL
```

- **Initialization**: *left = 1, right = n,* the invariant holds
  - Because there are no elements in *A* neither to the left of *left* nor to the right of *right*

Example: Binary Search (2)

- We want to make sure that if NIL is return *q* is not in A
- **Invariant**: *at the start of each **while** loop, q > A[i] for all i ∈[1..left-1] and q < A[i] for all i ∈[right+1..n]*

```
left←1
right←n
do
   j←⌊(left+right)/2⌋
   if A[j]=q then return j
   else if A[j]>q then right←j-1
   else left=j+1
while left<=right
return NIL
```

- **Maintenance**: if q < *A*[*j*], then q <*A*[*i*] for each *i* ∈[*j..n*]
  - Because the array is sorted, the algorithm assigns *j-1* to *right* (the second part of the invariant holds)
    - The first part of the invariant could similarly be shown to hold

Example: Binary Search (3)

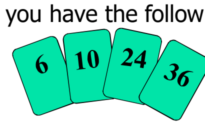
- We want to make sure that if NIL is return *q* is not in A
- **Invariant**: *at the start of each **while** loop, q > A[i] for all i ∈[1..left-1] and q < A[i] for all i ∈[right+1..n]*

```
left←1
right←n
do
   j←⌊(left+right)/2⌋
   if A[j]=q then return j
   else if A[j]>q then right←j-1
   else left=j+1
while left<=right
return NIL
```

- **Termination**: the loop terminates when *left > right*
  - The invariant states that *q* is smaller than all elements of *A* to the left of *left* and larger than all elements of *A* to the right of *right*
    - This covers all elements of *A*, i.e. *q* is either smaller or larger that any element of *A*

Insertion Sort

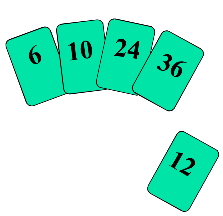
- One of the simplest sorting algorithms
  - Is not the simplest
  - People use it to sort cards in their hands
  - E.g. Suppose you have the following cards in your hand:



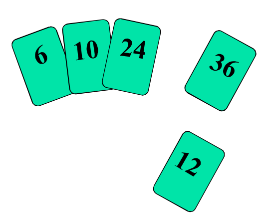
  - And received the card

Example: Insertion Sort

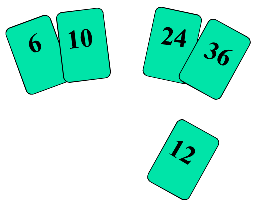
To insert 12, it is necessary to make room for this card by moving first 36 and then 24



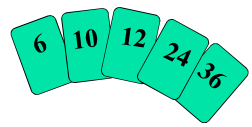
Example: Insertion Sort



Example: Insertion Sort



Example: Insertion Sort



Insertion Sort

Input array

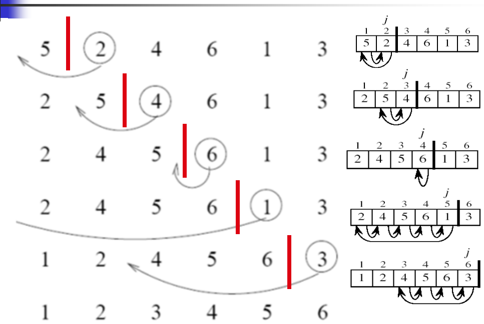
5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:

left sub-array | right sub-array
2 5 | 4 6 1 3

Sorted | Unsorted

Insertion Sort



Insertion Sort

6 5 3 1 8 7 2 4

https://upload.wikimedia.org/wikipedia/commons/0/0f/Insertion-sort-example-300px.gif

Example: Insertion Sort (1)

- **Invariant**: *at the start of each **for** loop, the elements in A[1…j-1] are in sorted order*

```
for j=2 to length(A)
  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)

- **Invariant**: *at the start of each **for** loop, the elements in A[1…j-1] are in sorted order*

```
for j=2 to length(A)
  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**: $j = 2$, the invariant trivially holds because $A[1]$ is a sorted array

Example: Insertion Sort (3)

- **Invariant**: *at the start of each **for** loop, the elements in A[1…j-1] are in sorted order*

```
for j=2 to length(A)
  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**: the **while** loop moves elements $A[j-1]$, $A[j-2]$, …, $A[j-k]$ one position to the right without changing their order
  - Then the former $A[j]$ element is inserted into k-th position so that $A[k-1] \leq A[k] < A[k+1]$
    - $A[1…j-1]$ sorted + $A[j]$ → $A[1…j]$ sorted

Example: Insertion Sort (4)

- **Invariant**: *at the start of each **for** loop, the elements in A[1…j-1] are in sorted order*

```
for j=2 to length(A)
  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**: the loop terminates, when $j=n+1$. Then the invariant states: *"A[1…n] consists of elements originally in A[1…n] but in sorted order"*

Asymptotic analysis

- Goal:
  - Simplify analysis of running time by ignoring "details" that may be affected by specific implementation and hardware
    - Like "rounding" for numbers: 1,000,001 ≈ 1,000,000
    - "Rounding" for functions: $3n^2 \approx n^2$
- Captures the essence:
  - How the running time of an algorithm increases with the size of the input *in the limit*
  - Algorithms asymptotically more efficient are the best for all but small inputs
- Written using asymptotic notation

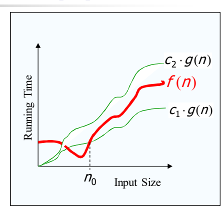
Asymptotic notation

- For Θ, ***O***, Ω, ***o***, ω
- Defined for functions over the natural numbers.
  - E.g.: $f(n) = \Theta(n^2)$.
  - Describes how $f(n)$ grows in comparison to n^2
- Define a ***set*** of functions
  - In practice used to compare two function sizes
- Describe different rate-of-growth relations between a defining function and a defined set of functions

Asymptotic notation (1)

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n) : \exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$, we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ \}



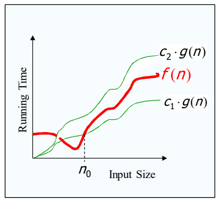
Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

$g(n)$ is an *asymptotically tight bound* for $f(n)$

Asymptotic notation (1)

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n))$ = {$f(n)$: $\exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$, we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ }



Technically, $f(n) \in \Theta(g(n))$
Old use, $f(n) = \Theta(g(n))$
Both can be used in this course

$f(n)$ and $g(n)$ are nonnegative, for large n

Example

$\Theta(g(n))$ = {$f(n)$: $\exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$}

- $10n^2 - 3n = \Theta(n^2)$
- What constants for n_0, c_1, and c_2 will work?
- Make c_1 a little smaller than the leading coefficient, and c_2 a little bigger.
  - *To compare orders of growth, look at the leading term*
- Exercise: Prove that $n^2/2-3n= \Theta(n^2)$

Example

$\Theta(g(n))$ = {$f(n)$: $\exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$}

- Is $3n^3 \in \Theta(n^4)$?
- How about $2^{2n} \in \Theta(2^n)$?

Asymptotic Notation

- Simple Rule: Drop lower order terms and constant factors
  - $50\ n \log n$ is ...
  - $7n - 3$ is ...
  - $8n^2 \log n + 5n^2 + n$ is ...

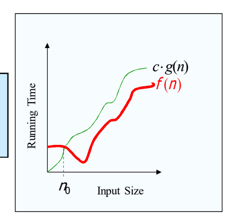
Asymptotic Notation

- 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 $O(n^2 \log n)$

Asymptotic Notation (2)

For a function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n))$ = {$f(n)$: $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)$ }



Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

$g(n)$ is an *asymptotic upper bound* for $f(n)$

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$
$\Theta(g(n)) \subset O(g(n))$

Example

$O(g(n)) = \{f(n) : \exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)$ $\}$

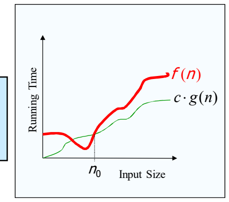
- Any linear *function an + b* is in $O(n^2)$?
  - Why?
- Show that $3n^3 = O(n^4)$ $(3n^3 \in O(n^4))$ for appropriate values of c and n_0

© André de Carvalho - ICMC/USP 43

Asymptotic Notation (3)

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$\Omega(g(n)) = \{f(n) : \exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq cg(n) \leq f(n)\}$



Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$

$g(n)$ is an *asymptotic lower bound* for $f(n)$

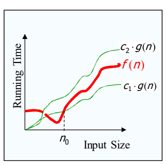
$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$

$\Theta(g(n)) \subset \Omega(g(n))$

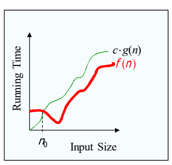
© André de Carvalho - ICMC/USP 44

Relations Between Θ, O, Ω

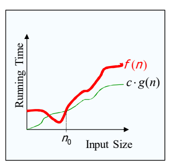
- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- $O(f(n))$ is often misused instead of $\Theta(f(n))$



$f(n) = \Theta(g(n))$



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



$f(n) = \Omega(g(n))$

© André de Carvalho - ICMC/USP 45

Asymptotic Notation (5)

- "Little-Oh" notation $f(n)=o(g(n))$ non-tight analogue of Big-Oh

  $o(g(n)) = \{f(n): \forall\ c > 0, \exists\ n_0 > 0$ such that $\forall\ n \geq n_0$, we have $0 \leq f(n) < cg(n)$ $\}$

  - Used for **comparisons** of running times
  - If $f(n)=o(g(n))$, it is said that *g(n) dominates f(n)*

Big-Oh $O(g(n)) = \{f(n) : \exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)$ $\}$

© André de Carvalho - ICMC/USP 46

Asymptotic Notation (6)

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

  $\omega(g(n)) = \{f(n): \forall\ c > 0, \exists\ n_0 > 0$ such that $\forall\ n \geq n_0$, we have $0 \leq f(n) < cg(n)$ $\}$

Big-Omega $\Omega(g(n)) = \{f(n) : \exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq cg(n) \leq f(n)\}$

© André de Carvalho - ICMC/USP 47

Asymptotic properties

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

© André de Carvalho - ICMC/USP 48

Limits

- $\lim_{n\to\infty} [f(n) / g(n)] = 0 \Rightarrow f(n) \in o(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] = \infty \Rightarrow f(n) \in \omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)]$ undefined $\Rightarrow$ Not possible to say

© André de Carvalho - ICMC/USP 49

Properties

- Symmetry

  $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$
- Transitivity

  $f(n) = \Theta(g(n))$ & $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$

  $f(n) = O(g(n))$ & $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

  $f(n) = \Omega(g(n))$ & $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$

  $f(n) = o(g(n))$ & $g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$

  $f(n) = \omega(g(n))$ & $g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$

© André de Carvalho - ICMC/USP 50

Properties

- Reflexivity

  $f(n) = \Theta(f(n))$

  $f(n) = O(f(n))$

  $f(n) = \Omega(f(n))$
- Complementarity

  $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

  $f(n) = o(g(n))$ iff $g(n) = \omega((f(n))$

© André de Carvalho - ICMC/USP 51

Brief Mathematical review

© André de Carvalho - ICMC/USP 52

Monotonicity

- $f(n)$ is
  - **monotonically increasing** if $m \leq n \Rightarrow f(m) \leq f(n)$
  - **monotonically decreasing** if $m \geq n \Rightarrow f(m) \geq f(n)$
  - **strictly increasing** if $m < n \Rightarrow f(m) < f(n)$
  - **strictly decreasing** if $m > n \Rightarrow f(m) > f(n)$

© André de Carvalho - ICMC/USP 53

Exponentials and Logarithms

- Properties of logarithms:

  $\log_b(xy) = \log_b x + \log_b y$

  $\log_b (x/y) = \log_b x - \log_b y$

  $\log_b xa = a\log_b x$

  $\log_b a = \log_x a/\log_x b$
- Properties of exponentials:

  $a^{(b+c)} = a^b a^c$

  $a^{bc} = (a^b)^c$

  $a^b / a^c = a^{(b-c)}$

  $b = a^{\log_a b}$

  $b^c = a^{c*\log_a b}$

© André de Carvalho - ICMC/USP 54

Bases of logarithms and exponentials

- The base of a logarithm can be changed multiplying the logarithm by a constant
  - E.g. $\log_{10} n * \mathbf{\log_2 10} = \log_2 n$
  - Base of logarithm is not important in asymptotic notation
- Exponentials with different bases differ by a exponential (not a constant)
  - E.g. $2^n = \mathbf{(2/3)^n} * 3^n$

Summations

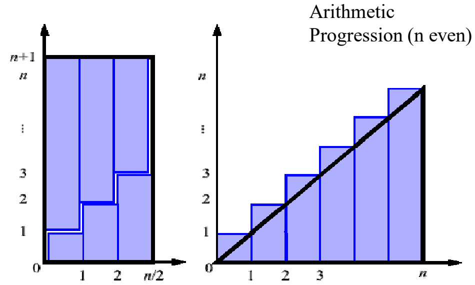
- Constant series
  - Given integers a and n, $a \leq n$

$$\sum_{i=a}^{n} 1 = n - a + 1$$

- Arithmetic progression (linear series)
  - Given an integer n

$$\sum_{i=0}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n(1+n)}{2}$$

Summations



Arithmetic Progression (n even)

Summations

- Quadratic Series
  - Given an integer $n \geq 0$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

- Cubic Series
  - Given an integer $n \geq 0$

$$\sum_{i=1}^{n} i^3 = 1^3 + 2^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$$

Summations

- Geometric progression (series)
  - Given an integer n and a real number $0 < a \neq 1$

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}$$

  - Geometric progressions exhibit exponential growth behaviour
  - For $|a| < 1$

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$$

Summations

- Linear-Geometric Series
  - Given an integer $n \geq 0$ and a real $c \neq 1$

$$\sum_{i=1}^{n} ic^i = c + 2c^2 + \cdots + nc^n = \frac{-(n+1)c^{n+1} + nc^{n+2} + c}{(c-1)^2}$$

- Harmonic Series
  - *Given a n^{th}* harmonic number, $n \in \mathrm{I}^+$

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \leq \sum_{i=1}^{n} \frac{1}{i} = \ln(n) + 1$$

Popular with architects, mainly in the Baroque period, to define Harmonic relations between interior and exterior architecture of churches and palaces

Summations

- The running time of insertion sort is determined by a nested loop

```
for j←2 to length(A)
   key←A[j]
   i←j-1
   while i>0 and A[i]>key
      A[i+1]←A[i]
      i←i-1
   A[i+1]←key
```

- Nested loops correspond to summations

$\sum_{j=2}^{n}(j-1)=O(n^2)$

© André de Carvalho - ICMC/USP 61

Proof by Induction

- Correctness estimation and time complexity estimation can be proved by mathematical induction
- Important mathematical tool for proofs
- Allow simple proofs

© André de Carvalho - ICMC/USP 62

Proof by Induction (1)

- 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*
- Example $S(n)=\sum_{i=0}^{n} i=\frac{n(n+1)}{2}$ for $n \geq 1$
- Basis $S(1)=\sum_{i=0}^{1} i=\frac{1(1+1)}{2}$

© André de Carvalho - ICMC/USP 63

Proof by Induction (2)

- Inductive Step

$$S(k)=\sum_{i=0}^{k} i=\frac{k(k+1)}{2} \text{ for } 1 \leq k \leq n-1$$

$$S(n)=\sum_{i=0}^{n} i=\sum_{i=0}^{n-1} i+n=S(n-1)+n=$$

$$=(n-1)\frac{(n-1+1)}{2}+n=\frac{(n^2-n+2n)}{2}=$$

$$=\frac{n(n+1)}{2}$$

© André de Carvalho - ICMC/USP 64

Next Week

- Divide-and-conquer
- Merge sort
- Writing recurrences to describe the running time of divide-and-conquer algorithms

© André de Carvalho - ICMC/USP 65

Acknowledgement

- A large part of this material were adapted from
  - Simonas Šaltenis, Algorithms and Data Structures, Aalborg University, Denmark
  - Mary Wootters, Design and Analysis of Algorithms, Stanford University, USA
  - George Bebis, Analysis of Algorithms CS 477/677, University of Nevada, Reno
  - David A. Plaisted, Information Comp 550-001, University of North Carolina at Chapel Hill

© André de Carvalho - ICMC/USP 66

Questions

© André de Carvalho - ICMC/USP 67