Introduction to Computers and Programming

Prof. I. K. Lundqvist

Lecture 9
April 7 2004

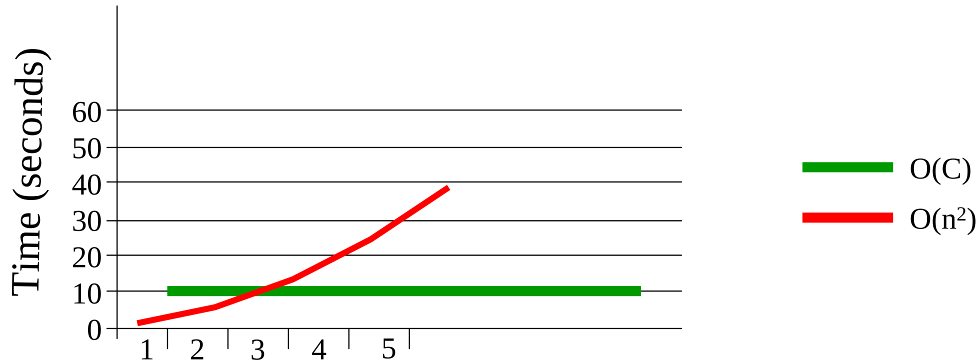
So far ...

- Data structures
- Algorithms

Complexity Analysis

"Just how good is my algorithm?"

- Best case vs. worst case
- Storage vs. Computation time
- Computing the computation time
- Big-O notation



In-class Exercise

- Write a procedure that reads an **integer N** and calculates the **sum of all integers 1..N**

Code Comparison

- How many have a solution that runs in linear time?

```
with Ada.Integer_Text_Io, Ada.Text_Io;
use Ada.Integer_Text_Io, Ada.Text_Io;

procedure CalcSum is
   N         : Integer;
   Total_Sum : Integer;
begin
   Put_Line("Enter an Integer: ");
   Get(N);
   Total_Sum := 0;
   for I in 1..N loop
      Total_Sum := Total_Sum + I;
   end loop;
   Put(Total_Sum);
end;
```


Code Comparison

- How many have a solution that runs in constant time? $\frac{N*(N+1)}{2}$

```
with Ada.Integer_Text_Io, Ada.Text_Io;
use Ada.Integer_Text_Io, Ada.Text_Io;

procedure Calcsum is
   N         : Integer;
   Total_Sum : Integer;

begin
   Put_Line("Enter an Integer: ");
   Get(N);
   Total_Sum := 0;
   Total_Sum := (N * (N + 1)) / 2;
   Put(Total_Sum);
end;
```

Complexity Analysis

- Complexity: rate at which storage or time grows as a function of the problem size
  - Growth depends on compiler, machine, …

- Asymptotic analysis: describes the inherent complexity of a program, independent of machine and compiler
  - **Idea**: as problem size grows, the complexity can be described as a simple proportionality to some known function.

Common Notations for Big-O

- O(1)

constant time or space

- O(N)
- O(log N)
- $O(N^M)$
- $O(M^N)$

Or a combination of these

O(1)

- Constant time or space, independently of what input we give to the algorithm

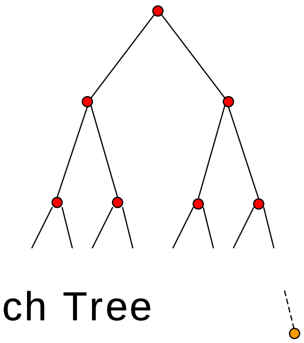
- Examples:
  - Access element in an array
  - Retrieve the first element in a list
  - ...

O(N)

- We have to search through all existing elements to find that the element we are looking for does not exist

- Examples:
  - Searching for element in a list that does not exist
  - Searching through a Binary Tree of size N where a value does not exist

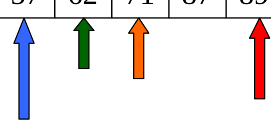
O(log N)



- Example, a full balanced Binary Search Tree
- Can eliminate half of the BST every time the search
- Any algorithm that eliminates a large portion of the data set at each iteration is generalized into O(log N)

Binary Search

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10	11	14	17	21	33	55	57	62	71	87	89	91	93	95	97



How many elements are examined in worst case?

Binary Search

```
Input:
    Array to search, element to search for
Output:
    Index if element found, -1 otherwise
Algorithm:
    Set Return_Index to -1;
    Set Current_Index to (UB + LB) /2

   Loop
     if the LB > UB
            Exit;

      if Input_Array(Current_Index) = element
            Return_Index := Current_Index
            Exit;

      if Input_Array(Current_Index) < element
         LB := Current_Index +1
     else
         UB := Current_Index - 1

  Return Return_Index
```


$O(N^M)$

```
 N := 1;
   while N > 0 loop
      Put("How many repetions? ");
      Get(N);
      X := 0;

      for I1 in 1..N loop
         for I2 in 1..N loop
            for I3 in 1..N loop
               for I4 in 1..N loop
                  for I5 in 1..N loop
                     X := X + 1;
                  end loop;
               end loop;
            end loop;
         end loop;
      end loop;
      Put(X);
      New_Line;
end loop;
```

$O(M^N)$

- Example: Fibonacci algorithm
  - $f(0) = 1$
  - $f(1) = 1$
  - $f(n+2) = f(n) + f(n+1) \quad \forall\ n \geq 0$

2^N calculations

Big-O

- O(N+M)
  - Sequential and unrelated tasks
  - Ex: to find the smallest N_1 and largest N_2 number in a list and generate a new list of all the numbers in between N_1 and N_2
- O(N*M)
  - Nesting of tasks
  - Ex: initializing a n-by-m matrix

Asymptotic Analysis: Big-O

- Mathematical concept that expresses "how good" or "how bad" an algorithm is

Definition: **T**(n) = O(f(n)) – *"T of n is in Big-Oh of f of n"*
iff there are constants **c** and $\mathbf{n_0}$ such that:
$\mathbf{T}(n) \leq cf(n)$ for all $n \geq n_0$

Usage: The algorithm is in O(n^2) in [best, average, worst] case.

Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in less than $cf(n)$ steps in [best, average, worst] case.

Big-O is said to describe an "upper bound" on the complexity.

Big-O Examples

Finding value *X* in an array (average cost).

$\mathbf{T}(n) = c_s n/2$.

$\mathbf{T}(n) = O(f(n))$ iff
$\mathbf{T}(n) \leq cf(n)$ for all $n \geq n_0$

For all values of $n > 1$, $c_s n/2 <= c_s n$.

Therefore, by the definition, $\mathbf{T}(n)$ is in $O(n)$ for $n_0 = 1$ and $c = c_s$.

Big-O Example

$\mathbf{T}(n) = c_1n^2 + c_2n$ in average case.

$\mathbf{T}(n) = O(f(n))$ iff
$\mathbf{T}(n) \leq cf(n)$ for all $n \geq n_0$

$c_1n^2 + c_2n <= c_1n^2 + c_2n^2 <= (c_1 + c_2)n^2$ for all $n > 1$.

$\mathbf{T}(n) <= cn^2$ for $c = c_1 + c_2$ and $n_0 = 1$.

Therefore, $\mathbf{T}(n)$ is in $O(n^2)$ by the definition

Big-O Simplifications

O(2*N)	Same as	O(N)
$O(5*3^N)$	Same as	$O(3^N)$
O(4711)	Same as	O(1)
O(N+1)	Reduces to	O(N)
$O(N^2+\log N)$	Reduces to	$O(N^2)$
$O(N*\log N+2^N+50000)$	Reduces to	$O(2^N)$

Big-O Simplifications

$O(N+P+Q)$	Same as	$O(N+P+Q)$
$O(5*N^3 + 7N+ 2P+Q*R)$	Reduces to	$O(5*N^3+2P+Q*R)$
$O(N^2 \log P+N)$	Same as	$O(N^2 \log P+N)$
$O(N*M+N^2)$	Same as	$O(N*M+N^2)$

Faster Computer or Algorithm?

The old computer processes 10,000 instructions per hour

What happens when we buy a computer 10 times faster?

T(n)	n	n'	n'/n
$10n$	1,000	10,000	10
$20n$	500	5,000	10
$5n \log n$	250	1,842	7.37
$2n^2$	70	223	3.16
2^n	13	16	-----