Correctness/efficiency, runtime/space w.r.t. order notation, linear & binary search

Data structures & algorithms (COSC202A)

Lecture 2

Correctness

- Let’s study a problem to understand proving correctness
- We will consider the searching problem
- Suppose we are given a set of numbers:
- 5, 3, 10, 7, 13
- Can you find 7 in this list?

Linear search

ALGORITHM 2 The Linear Search Algorithm.

procedure *linear search*(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
        $i := i + 1$
if $i \leq n$ **then** $location := i$
else $location := 0$
{$location$ is the subscript of the term that equals x, or is 0 if x is not found}

Correctness

- First we will figure out a claim, proving which will show the correctness of the algorithm
- The complicated part here is the algorithm will provide the correct result at the end
- So how we are going to analyze the correctness of when the algorithm is in the middle of a computation?
- We need a claim that takes the algorithms steps into account
- Such claim is called a loop invariant
- For linear search an appropriate claim is “In i-th step the element is not present in the list of first i-1 elements”

Correctness

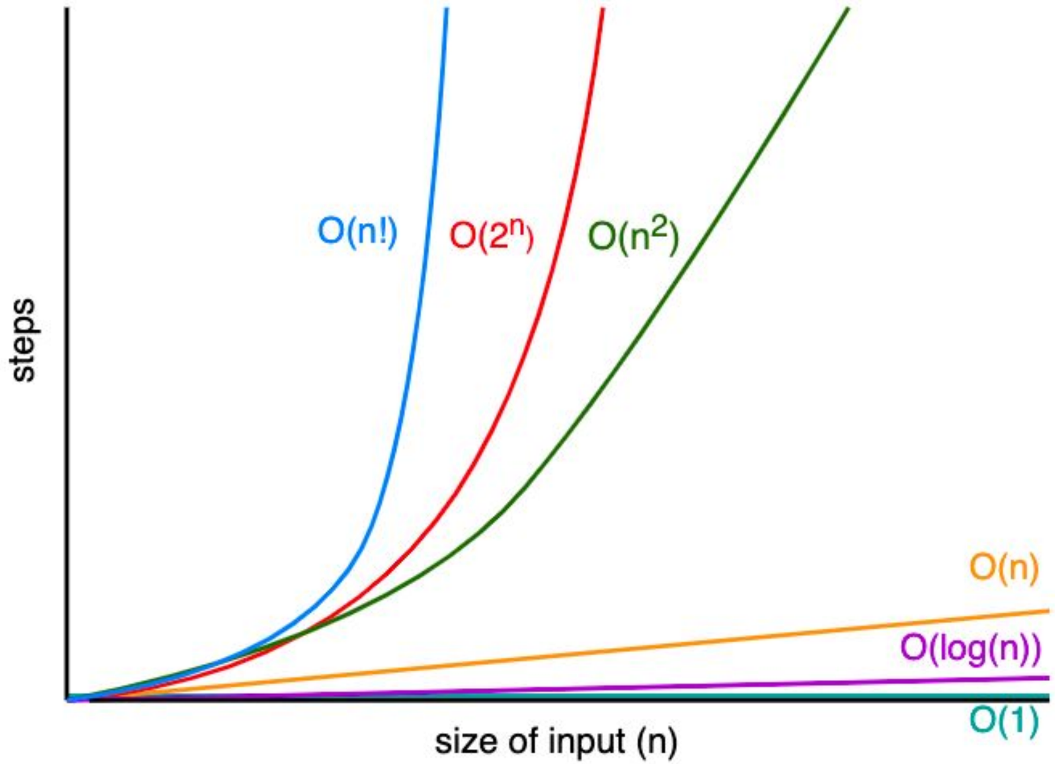
- Once we have a loop invariant, we need to show this three steps:
  - Initialization
  - Maintenance
  - Termination

Runtime/space

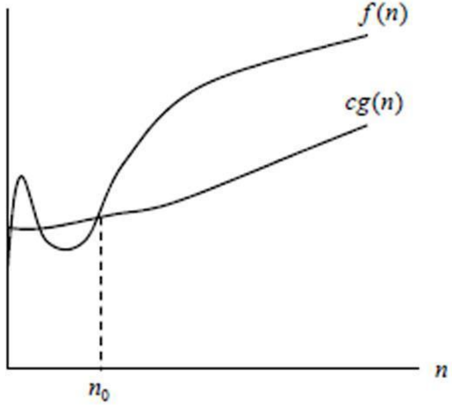
- For runtime/space analysis we will use asymptotic notations
- There are three major types of notations:
    - Order notation
    - Omega notation
    - Theta notation

O-notation

f(n) is O(g(n)) if there exists c and n_0 such that f(n) < c g(n) when $n>n_0$



The Ω Notation



$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$

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

Examples:

$n^2 = \Omega(n^2)$

$n^2 + n = \Omega(n^2)$

$2311n^2 + 1000n = \Omega(n^2)$

$n^3 = \Omega(n^2)$

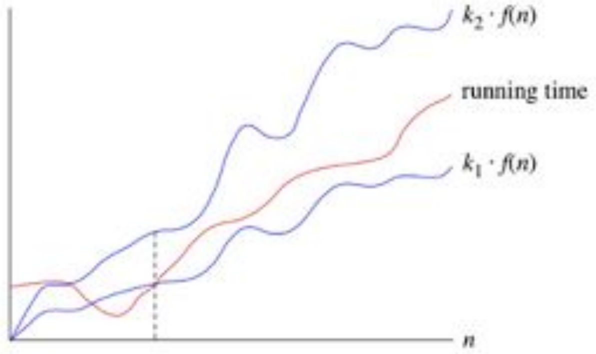
$n^{2.0001} = \Omega(n^2)$

$n^2 \lg\lg n = \Omega(n^2)$

$2^{2^n} = \Omega(n^2)$

Asymptotically tight bound

We are not restricted to just n in big-Θ notation. We can use any function, such as n^2, $n \lg n$, or any other function of n. Here's how to think of a running time that is $\Theta(f(n))$ for some function $f(n)$:



Once n gets large enough, the running time is between $k_1 \cdot f(n)$ and $k_2 \cdot f(n)$.

Monotonicity

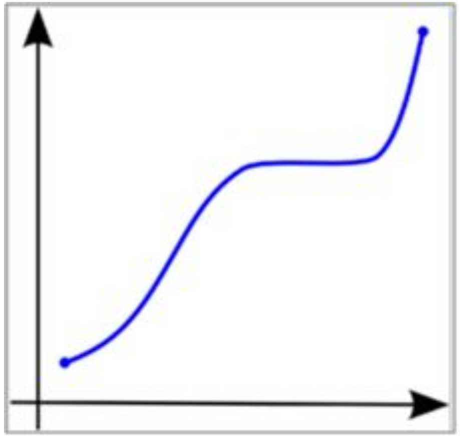


Figure 1 - A monotonically increasing function

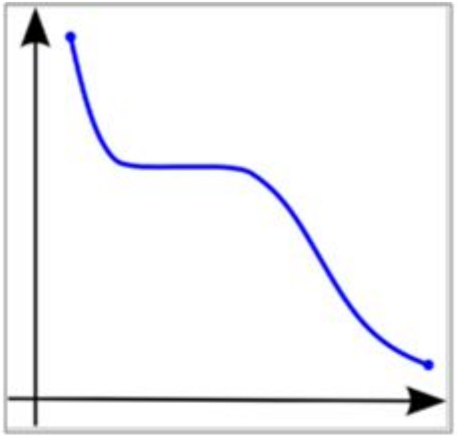


Figure 2 - A monotonically decreasing function

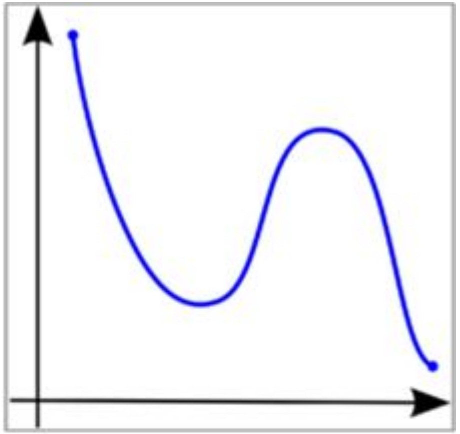
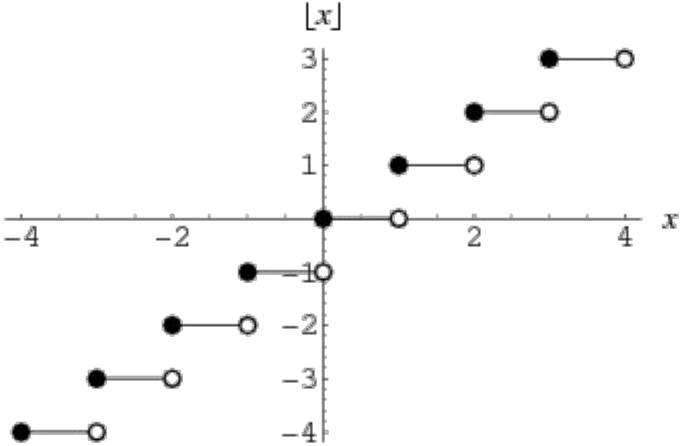
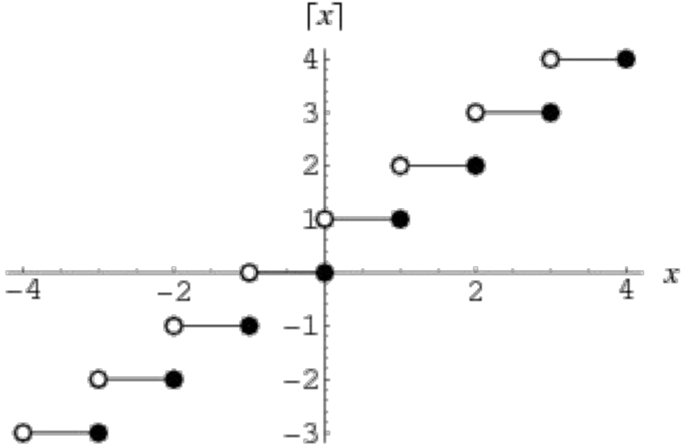


Figure 3 - A function that is not monotonic

Floors and ceilings

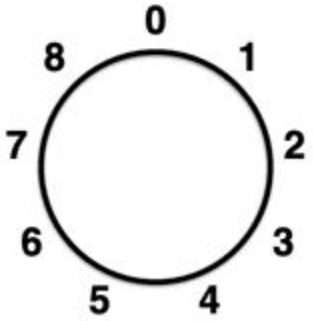




Modular arithmetic

Modulus 9

0 mod 9 = 0	9 mod 9 = 0
1 mod 9 = 1	10 mod 9 = 1
2 mod 9 = 2	11 mod 9 = 2
3 mod 9 = 3	12 mod 9 = 3
4 mod 9 = 4	13 mod 9 = 4
5 mod 9 = 5	14 mod 9 = 5
6 mod 9 = 6	15 mod 9 = 6
7 mod 9 = 7	16 mod 9 = 7
8 mod 9 = 8	17 mod 9 = 8

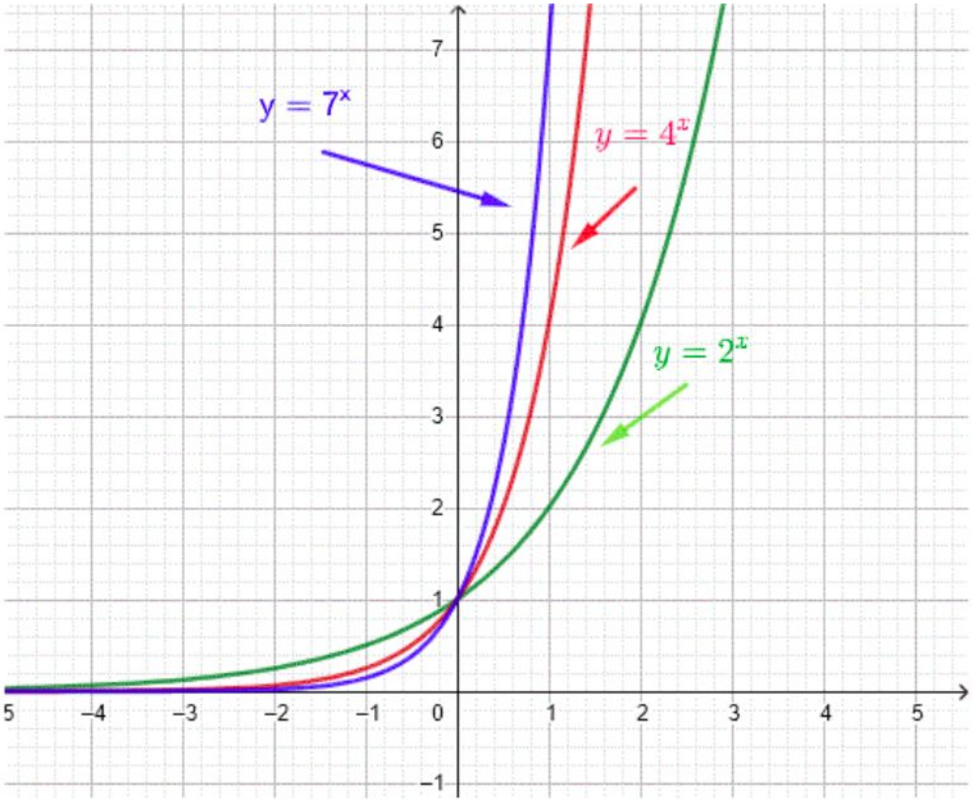


Polynomials

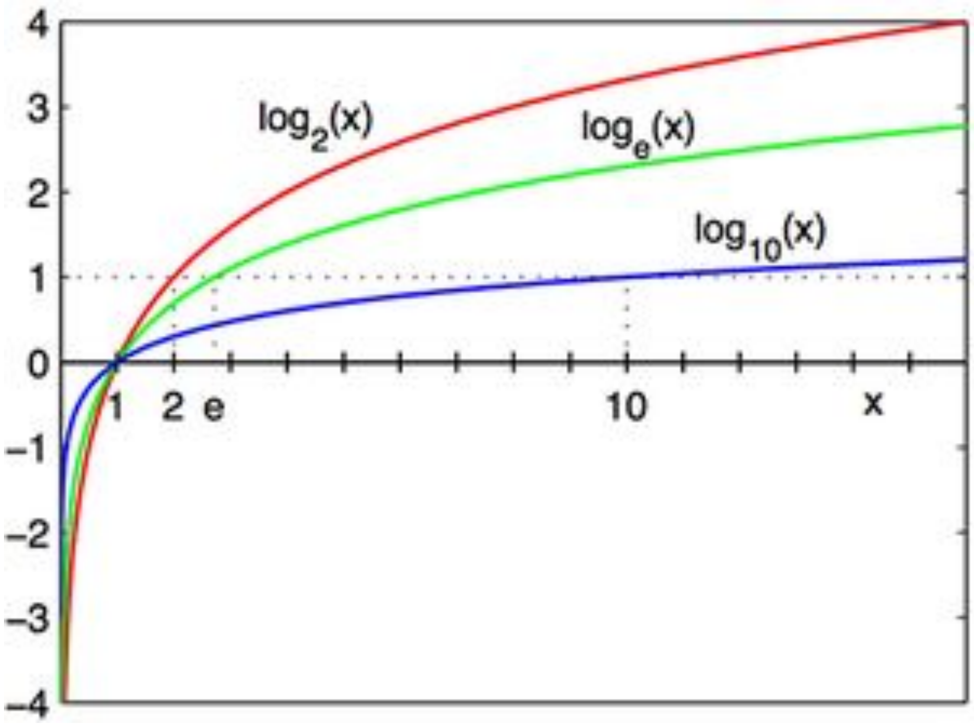
- Here *d* is the **degree** of the polynomial
- The term a_i is the **coefficient**
- Polynomials are very common in algorithmic analysis

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

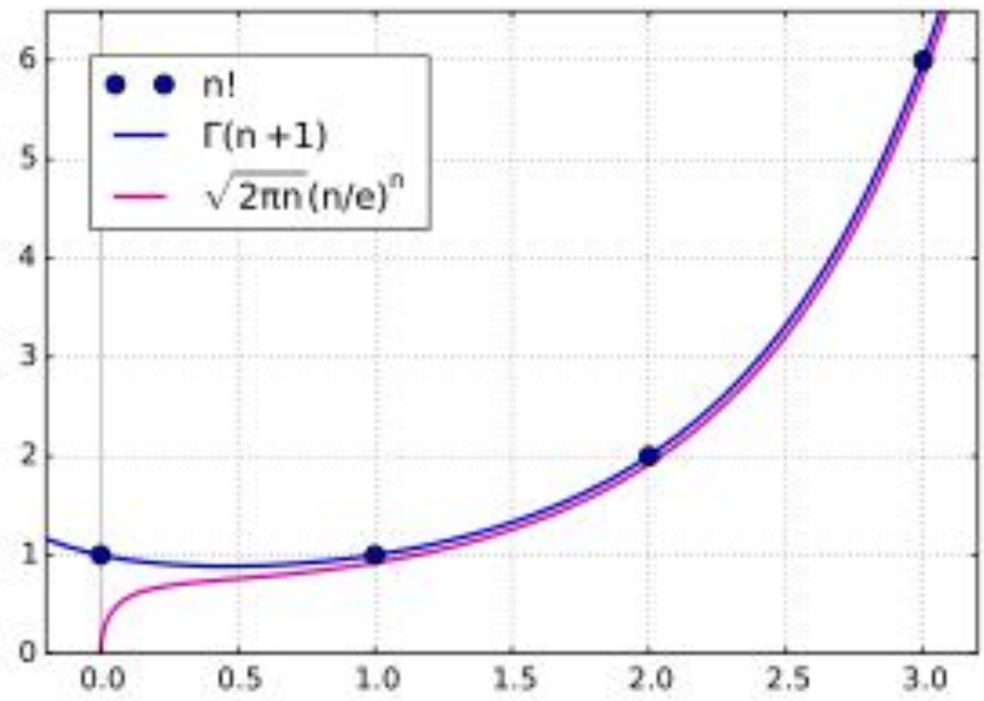
Exponentials



Logarithms



Factorials



Functional iteration

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0\,, \\ f(f^{(i-1)}(n)) & \text{if } i > 0\,. \end{cases}$$

For example, if $f(n) = 2n$, then $f^{(i)}(n) = 2^i n$.

The iterated logarithm function

The iterated logarithm is a *very* slowly growing function:

$$
\begin{aligned}
\lg^* 2 &= 1\,, \\
\lg^* 4 &= 2\,, \\
\lg^* 16 &= 3\,, \\
\lg^* 65536 &= 4\,, \\
\lg^*(2^{65536}) &= 5\,.
\end{aligned}
$$

The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2
1+2=3
2+3=5
3+5=8
5+8=13
8+13=21

13+21=34
21+34=55
34+55=89
55+89=144
89+144=233
144+233=377

Binary search

ALGORITHM 3 The Binary Search Algorithm.

procedure *binary search* (x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1$ {i is left endpoint of search interval}
$j := n$ {j is right endpoint of search interval}
while $i < j$
begin
  $m := \lfloor (i + j)/2 \rfloor$
  if $x > a_m$ **then** $i := m + 1$
  else $j := m$
end
if $x = a_i$ **then** *location* $:= i$
else *location* $:= 0$
{*location* is the subscript of the term equal to x, or 0 if x is not found}