Analysis of Algorithms I

Johannes Borgström
johannes.borgstrom@it.uu.se

Program Design and Data Structures

Based on notes by Tjark Weber and Dave Clarke

Overview

- Introduction to Analysis of Algorithms
- Growth of Functions
- Big Θ, O, and Ω Notations
- Recurrences
- From Code to Recurrences
- Examples

What is an Algorithm?

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

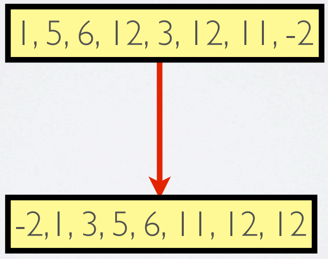
An algorithm is thus a sequence of computational steps that transform its input into its output.

An algorithm solves a computational problem, phrased in terms of a desirable input/output relationship.

The Sorting Problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$



Analysis of Algorithms

Different algorithms designed to solve the same problem differ dramatically in their efficiency.

These differences are more significant than differences due to hardware and software.

$\Rightarrow$ absolute costs are not the most important measures.

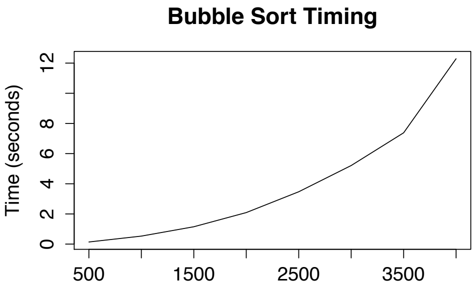
Algorithm analysis studies the cost of algorithms in terms of **time** and **space complexity**, ignoring constant factors.

Motivation: Bubble Sort: How Expensive is it?

```
bubbleSort :: Ord a => [a] -> [a]
bubbleSort s = case bubble s of
                 t | t == s -> t
                   | otherwise -> bubbleSort t
  where bubble (x:y:xs) | x > y = y:(bubble (x:xs))
        bubble s = s
```

Motivation: Bubble Sort: How Expensive is it?

```haskell
bubbleSort :: Ord a => [a] -> [a]
bubbleSort s = case bubble s of
                t | t == s -> t
                  | otherwise -> bubbleSort t
  where bubble (x:y:xs) | x > y = y:(bubble (x:xs))
        bubble s = s
```



Intuition about Complexity

Analyse an algorithm without running it to gain some understanding of its performance.

Can apply to algorithms — before the program is written.

Even if the analysis is approximate, performance problems may be detected.

Helpful in documenting software libraries — programs using such libraries can be analysed without requiring analysis of the library source code (often not available).

Intuition about Complexity

Different ways of performing analysis:

Worst-case running time — the longest run for an input of a given size

Best-case running time — the shortest run for an input of a given size.

Average-case running time — the average time over all inputs of a given size.

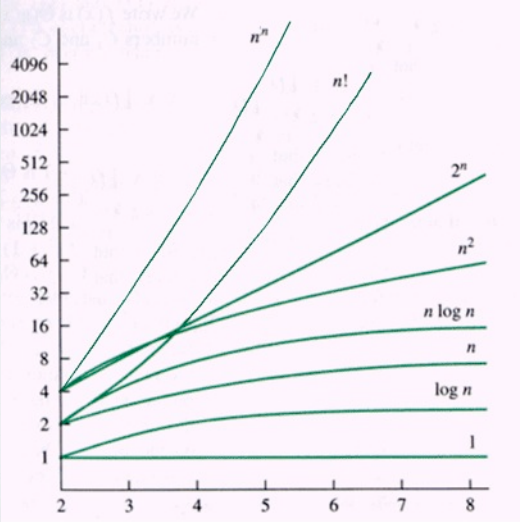
Similarly for space usage.

Overview of Techniques

Road map to what is ahead.

- Growth of functions, big O-, Ω-, Θ-notations
- Runtime equations (recurrences)
- Converting code to runtime equations
- Techniques for solving recurrences (next lecture and beyond)
  - Expansion method
  - Substitution method
  - Doctor Theorem

Growth of Functions



Growth of Functions

size of input

	10	**20**	**30**	**40**	**50**	**60**	**70**
log	1	1	1	2	2	2	2
linear	10	20	30	40	50	60	70
n log n	10	26	44	64	85	107	129
quadratic	100	400	900	1600	2500	3600	4900
cubic	1000	8000	27000	64000	125000	216000	343000
exponential	22026	485165195	10686474581525	235385266837020000	5184705528587070000000	114200738981568000000000000	2515438670919170000000000000000
log	1s	1s	1s	2s	2s	2s	2s
linear	10s	20s	30s	40s	50s	1m	1m 10s
n log n	10s	26s	44s	1m	1m 30s	2m	2m
quadratic	2m	7m	15m	27m	42m	1h	1h 22m
cubic	17m	2h 13m	7h 30m	17h 46m	1d 11h	2d 12h	3d 23h
exponential	6h	15y	338 865y	7 billion years	long time	longer time	longest time

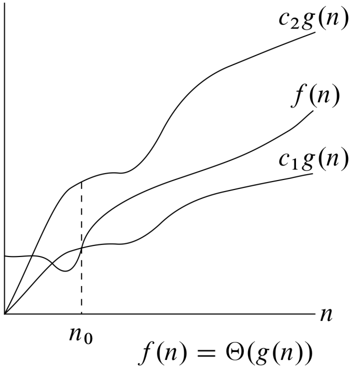
Terminology

Function	Growth Rate	
1	constant	sublinear
$\log n$	logarithmic	
$\log^2 n$	log squared	
n	linear	polynomial
$n \log n$		
n^2	quadratic	
n^3	cubic	
k^n	exponential	exponential
$n!$		superexponential
n^n		

n denotes the size of input. $k > 1$ is a constant.
$\log n$ is logarithm base 2, though the base doesn't really matter.

Big Θ Notation

The Θ notation is used to denote a **set** of functions that increase at the same rate (within some constant bound).



The Θ notation is used to give asymptotically **tight** bounds.

Big Θ Notation

The function $g(n)$ in $\Theta(g(n))$ is called a **complexity function.**

We sometimes write $f(n) = \Theta(g(n))$ to mean $f(n) \in \Theta(g(n))$.

Using '$=$' is traditional, but confusing. Think in terms of $\in$!!!

Definition

For non-negative functions f and g, $f(n) \in \Theta(g(n))$ if and only if there exist $n_0 \geq 0$ and c_1, $c_2 > 0$ such that for **all** $n > n_0$

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$$

$\Theta(g(n))$ is the set of all functions $f(n)$ that are bounded below and above by constant multiples of $g(n)$ when n gets sufficiently large.

Example

Theorem: $n^2 + 5n + 10 \in \Theta(n^2)$.

Proof: We need to find constants $n_0 \geq 0$ and c_1, $c_2 > 0$ such that

$$c_1 n^2 \leq n^2 + 5n + 10 \leq c_2 n^2$$

for all $n > n_0$. Dividing by n^2 (assuming $n > 0$) gives

$$c_1 \leq 1 + \frac{5}{n} + \frac{10}{n^2} \leq c_2$$

The term $1 + \frac{5}{n} + \frac{10}{n^2}$ gets smaller as n grows. It peaks at 16 for $n = 1$, so we can pick $c_2 = 16$.

It approaches 1 as $n \to \infty$, but is never less than 1. So we can pick $c_1 = 1$.

These choices of c_1 and c_2 work for any $n > 0$, so we can pick $n_0 = 0$.

Keeping Complexity Functions Simple

While it is formally correct to say

$$4n^2 + 5n + 10 = \Theta(4n^2 + 5n + 10)$$

the whole purpose of the Θ notation is to work with simpler expressions.

Write instead

$$4n^2 + 5n + 10 = \Theta(n^2)$$

Rule of thumb for polynomials: Set constant factors to 1 and drop lower-order terms.

Complexity of Built-in Functions

Let $|D|$ denote the number of elements in a data structure D.

Builtin Function	Time Complexity
pattern matching	$\Theta(1)$ time always
a {`+`,`-`,`*`,`/`,`div`} b	$\Theta(1)$ time always*
min/max a b	$\Theta(1)$ time always*
$h : T$	$\Theta(1)$ time always
L ++ R	$\Theta(\lvert L \rvert)$ time always
`length` L	$\Theta(\lvert L \rvert)$ time always
a {`<`,`<=`,`==`,`>=`,`>`} b	$\Theta(1)$ time always*
`reverse` L	$\Theta(\lvert L \rvert)$ time always

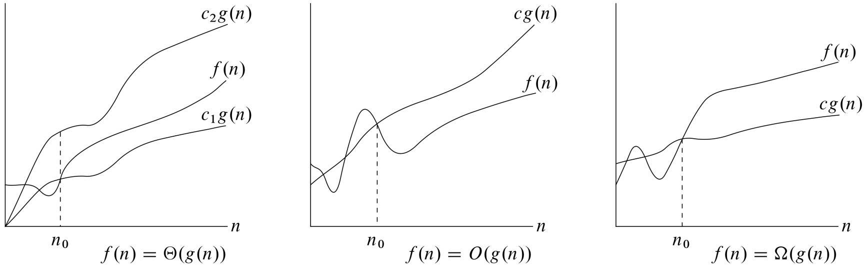
*For `Int` and other primitive types—simplifying practical assumption.

Variations on Big Θ: Big O and Ω Notations

Big Θ is a tight bound.

Big O is an upper bound only.

Big Ω is a lower bound only.



Variations on Big Θ: Big O and Ω Notations

Definition

For non-negative functions f and g, $f(n) \in O(g(n))$ if and only if there exist $n_0 \geq 0$ and $c > 0$ such that for **all** $n > n_0$ we get $f(n) \leq c \cdot g(n)$.

$O(g(n))$ is the set of all functions $f(n)$ that are bounded above by a constant multiple of $g(n)$ when n gets sufficiently large.

Definition

For non-negative functions f and g, $f(n) \in \Theta(g(n))$ if and only if there exist $n_0 \geq 0$ and $c > 0$ such that for **all** $n > n_0$ we get $c \cdot g(n) \leq f(n)$.

$\Theta(g(n))$ is the set of all functions $f(n)$ that are bounded below by a constant multiple of $g(n)$ when n gets sufficiently large.

Variations on Big Θ: Big O and Ω Notations (Example)

Any quadratic function $an^2 + bn + c$ is in $\Theta(n^2)$, and hence in $O(n^2)$. It is also in $O(n^3)$, $O(2^n)$, $O(n!)$, ... but *not* in $\Theta(n^3)$, $\Theta(2^n)$, $\Theta(n!)$, ...

Moreover, it is in $\Omega(n^2)$. It is also in $\Omega(n)$, $\Omega(1)$, ... but *not* in $\Theta(n)$, $\Theta(1)$, ...

It is *not* in $O(n)$. It is also *not* in $\Omega(n^3)$.

Recurrences

A **recurrence** (or **recurrence relation**) is an equation that recursively defines a sequence in terms of 1) one or more initial terms and
2) a function defined in terms of the preceding terms.

For example:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2), \quad \text{if } n > 1 \end{aligned}$$

Defines the sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Recurrences can be used to describe the runtime of functions.

Code to Recurrences

The following function calculates the sum of a list of integers:

```
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

Assume:

- + takes constant time, say t_{add}.
- Pattern matching (`[]`) takes constant time, say t_0.
- Pattern matching (`x:xs`) takes constant time, say t_1.

Hence, only the length of the list (but not its contents) matters for cost.

Runtime cost $T(n)$ is defined by this recurrence:

$$T(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T(n-1) + t_{add} + t_1 & \text{if } n > 0 \end{cases}$$

Closed Form

The closed form solution of the previous equation is

$$T(n) = t_0 + (t_{add} + t_1) \cdot n$$

This is a useful predictor of the actual runtime of `sumList`.

Even if t_0, t_1 and t_{add} were measured accurately, the actual runtime would vary with every change in the hardware or software environment. Therefore, the actual values of these constants are not of interest!

Rather

$$T(n) \in \Theta(n)$$

E.g., calling `sumList` with a list twice as long will double the runtime.

Problem: deriving such closed forms can be difficult (see next lecture).

Example: head

Consider the function

```
head :: [a] -> a
head [] = error "head applied to empty list"
head (a : _) = a
```

Runtime cost $T(n)$, where n is the length of the list, is defined by this recurrence:

$$
\begin{aligned}
T(n) &= \begin{cases} t_0 & \text{if } n = 0 \\ t_1 & \text{if } n > 0 \end{cases} \\
&= \Theta(1)
\end{aligned}
$$

Example: length

Consider the function

```haskell
length :: [a] -> Int
length [] = 0
length (_ : l) = 1 + length l
```

Runtime cost $T(n)$, where n is the length of the list, is defined by this recurrence:

$$T(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T(n-1) + t_{add} + t_1 & \text{if } n > 0 \end{cases}$$

Closed form: $T(n) \in \Theta(n)$.

Example: Complicated

Consider this function, which takes a list of points and returns the distance (squared) between the closest pair of points:

```
type Point = (Int, Int)

closestPair :: [Point] -> Int
closestPair (p : q : []) = distance p q
closestPair (p : q : l) =
  min (closestPair' (distance p q) p l) (closestPair (q : l))

closestPair' :: Int -> Point -> [Point] -> Int
closestPair' closest p [] = closest
closestPair' closest p (q : l) =
  closestPair' (min closest (distance p q)) p l

distance (x1, y1) (x2, y2) = (x1 - x2) ^ 2 + (y1 - y2) ^ 2
```

(This is not the best possible algorithm.)

Calculations

```
distance (x1, y1) (x2, y2) = (x1 - x2) ^ 2 + (y1 - y2) ^ 2
```

Runtime cost of `distance` is $\Theta(1)$.

Calculations

```
closestPair' closest p [] = closest
closestPair' closest p (q : l) =
  closestPair' (min closest (distance p q)) p l
```

Runtime cost of `closestPair'` is

$$T(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T(n-1) + t_1 & \text{if } n > 0 \end{cases}$$

Closed form: $T(n) \in \Theta(n)$.

Calculations

```
closestPair (p : q : []) = distance p q
closestPair (p : q : l) =
  min (closestPair' (distance p q) p l) (closestPair (q : l))
```

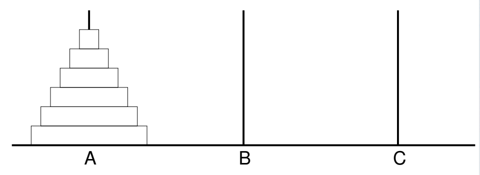
Runtime cost of `closestPair` is

$$C(n) = \begin{cases} t_0 & \text{if } n = 2 \\ T(n-2) + C(n-1) + t_1 & \text{if } n > 2 \end{cases}$$

Closed form: $C(n) \in \Theta(n^2)$.

Example: Tower of Hanoi

Initial state: Tower *A* has *n* disks stacked by decreasing diameter. Towers *B* and *C* are empty.



Rules

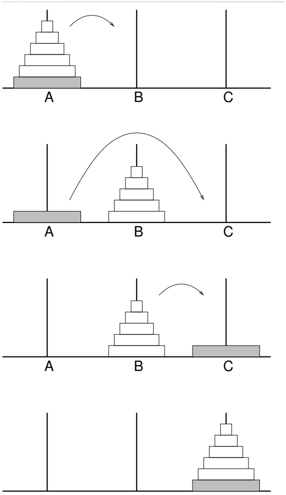
- Only move one disk at a time.
- Only move the top-most disk of a tower
- Only move a disk onto a larger disk (if any).

Objective: Move all disks from tower *A* to tower *C* using tower *B* without violating the rules.

Problem: What is the (minimal) sequence of moves?

Example: Tower of Hanoi: Strategy

1. Recursively move $n-1$ disks from tower A to B using C.
2. Move one disk from A to C
3. Recursively move $n-1$ disks from B to C using A.



Example: Tower of Hanoi: Specification and Code

```
{- hanoi n from via to
   PRE: n>=0
   RETURNS: description of the moves to be made for transferring n disks from
        tower `from` to tower `to`, using tower `via`
 -}
-- VARIANT: n
hanoi :: Int -> String -> String -> String -> String
hanoi 0 from via to = ""
hanoi n from via to =
    hanoi (n-1) from to via ++
    from ++ "->" ++ to ++ " " ++
    hanoi (n-1) via from to
```

Will the following call finish before the end of the universe?

```
hanoi 64 "A" "B" "C"
```

Example: Tower of Hanoi: Analysis

```
hanoi 0 from via to = ""
hanoi n from via to =
    hanoi (n-1) from to via ++
    from ++ "->" ++ to ++ " " ++
    hanoi (n-1) via from to
```

Let $M(n)$ be the **number** of moves that must be made for solving the problem of the Towers of Hanoi with n disks (using the above strategy).

From the program, we get the recurrence

$$M(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 \cdot M(n-1) + 1 & \text{if } n > 0 \end{cases}$$

Closed form: $M(n) \in \Theta(2^n)$.

Summary

- Introduction to Analysis of Algorithms
- Growth of Functions
- Big Θ, O, and Ω Notations
- Recurrences
- From Code to Recurrences
- Examples