

# Introduction - Review

## Advanced Algorithms and Data Structures - Lecture 1

---

Venanzio Capretta

Thursday 1 October 2020

School of Computer Science, University of Nottingham

# The Maximum Subarray Problem

---

# Maximum Subarray Example

Consider the following list of integers:

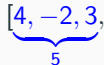
$[4, -2, 3, -7, 5, 2, -6, 8, -4, 3, -2, 1]$

Problem: Find the sublist with maximum sum

# Maximum Subarray Example

Consider the following list of integers:

[4, -2, 3, -7, 5, 2, -6, 8, -4, 3, -2, 1]



Some sublists:

start index	end index	sum
0	2	5

**Problem:** Find the sublist with maximum sum

# Maximum Subarray Example

Consider the following list of integers:

[4, -2, 3, -7, 5, 2, -6, 8, -4, 3, -2, 1]  
6

Some sublists:

start index	end index	sum
0	2	5
7	11	6

**Problem:** Find the sublist with maximum sum

# Maximum Subarray Example

Consider the following list of integers:

[4, -2, 3, -7, 5, 2, -6, 8, -4, 3, -2, 1]

7

Some sublists:

start index	end index	sum
0	2	5
7	11	6
4	5	7

**Problem:** Find the sublist with maximum sum

# Maximum Subarray Example

Consider the following list of integers:

[4, -2, 3, -7, 5, 2, -6, 8, -4, 3, -2, 1]

9

Some sublists:

start index	end index	sum
0	2	5
7	11	6
4	5	7

**Problem:** Find the sublist with maximum sum

In this case it is the list with:

start index = 4   end index = 7   sum = 9

# Maximum Subarray Example

Consider the following list of integers:

[4, -2, 3, -7, 5, 2, -6, 8, -4, 3, -2, 1]  
9

Some sublists:

start index	end index	sum
0	2	5
7	11	6
4	5	7

**Problem:** Find the sublist with maximum sum

In this case it is the list with:

start index = 4   end index = 7   sum = 9

**Write an algorithm to compute the maximum sublist**

(Applications: gene sequence analysis, computer vision, data mining)



# Brute Force Algorithm

We will look at several algorithms for the Maximum Subarray Problem

The first is a **brute force** algorithm that checks all sublists:

# Brute Force Algorithm

We will look at several algorithms for the Maximum Subarray Problem

The first is a **brute force** algorithm that checks all sublists:

- Create the list of all sublists

$[[4], [4, -2], [4, -2, 3], [4, -2, 3, -7], \dots, [3, -2, 1], [-2], [-2, 1], [1]]$

# Brute Force Algorithm

We will look at several algorithms for the Maximum Subarray Problem

The first is a **brute force** algorithm that checks all sublists:

- Create the list of all sublists

$[[4], [4, -2], [4, -2, 3], [4, -2, 3, -7], \dots, [3, -2, 1], [-2], [-2, 1], [1]]$

- Compute the sum of all sublists

$[4, 2, 5, -2, \dots, 2, -2, -1, 1]$

# Brute Force Algorithm

We will look at several algorithms for the Maximum Subarray Problem

The first is a **brute force** algorithm that checks all sublists:

- Create the list of all sublists

$[[4], [4, -2], [4, -2, 3], [4, -2, 3, -7], \dots, [3, -2, 1], [-2], [-2, 1], [1]]$

- Compute the sum of all sublists

$[4, 2, 5, -2, \dots, 2, -2, -1, 1]$

- Take the maximum

$\text{maximum}[4, 2, 5, -2, \dots, 2, -2, -1, 1] = 9$

# Brute Force Algorithm

We will look at several algorithms for the Maximum Subarray Problem

The first is a **brute force** algorithm that checks all sublists:

- Create the list of all sublists

$[4], [4, -2], [4, -2, 3], [4, -2, 3, -7], \dots, [3, -2, 1], [-2], [-2, 1], [1]$

- Compute the sum of all sublists

$[4, 2, 5, -2, \dots, 2, -2, -1, 1]$

- Take the maximum

$\text{maximum } [4, 2, 5, -2, \dots, 2, -2, -1, 1] = 9$

This is very inefficient (cubic complexity)

Small improvement: reuse the sums already computed

# Haskell Code

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0 ≥ s then (0,j0,s0)
              else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

# Haskell Code

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0 ≥ s then (0,j0,s0)
              else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

The **type** of the function `maxSub` says that

`maxSub` is a function that **maps a list to a triple  $(i, j, s)$**

- $i$  is the index of the first element of the sublist
- $j$  is the index of the last element of the sublist
- $s$  is the sum of the sublist

# Haskell Code

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0 ≥ s then (0,j0,s0)
              else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

If the input is a [singleton list](#) `[x]`

then the singleton is the maximum list

- 0: index of `x`, first element of the sublist
- 0: `x` is also the last element of the sublist
- `x`: the sum of `[x]` is `x`



# Haskell Code

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0 ≥ s then (0,j0,s0)
              else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

For longer lists we split the computation in two parts

- Sublists that contain the first element of `xs`  
let  $(0, j_0, s_0)$  be the maximum of them
- Sublists that do not contain the first element  
Then they are sublists of `(tail xs)`  
let  $(i, j, s)$  be the recursive max sublist of `(tail xs)`
- Choose the larger of the two

# Haskell Code

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0 ≥ s then (0,j0,s0)
              else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

## Auxiliary functions:

- `sums` computes the progressive sums of lists starting at the beginning  
`sums [1,-2,3,5] = [(0,1),(1,-1),(2,2),(3,7)]` because
  - the sum of [1] (indices 0 and 0) is 1  $\Rightarrow$  (0,1)
  - the sum of [1,-2] (indices 0 and 1) is -1  $\Rightarrow$  (1,-1)
  - the sum of [1,-2,3] (indices 0 and 2) is 2  $\Rightarrow$  (2,2)
  - the sum of [1,-2,3,5] (indices 0 and 3) is 7  $\Rightarrow$  (3,7)
- `argMax snd` selects the element with the maximum second component (the sum)

# Do I need to know Haskell?

Q: Am I supposed to program in Haskell?

# Do I need to know Haskell?

Q: Am I supposed to program in Haskell?

A: NO, you just need to understand the code

# Do I need to know Haskell?

Q: Am I supposed to program in Haskell?

A: NO, you just need to understand the code

- I give examples and solutions in Haskell  
You must be able to understand my code  
I only use basic Haskell (no Advanced Functional Programming)

# Do I need to know Haskell?

Q: Am I supposed to program in Haskell?

A: NO, you just need to understand the code

- I give examples and solutions in Haskell  
You must be able to understand my code  
I only use basic Haskell (no Advanced Functional Programming)
- You don't need to program in Haskell yourself  
Use your favourite programming language  
The textbook has pseudocode in imperative style

# Complexity of the Algorithm

Exercise: What is the complexity of the `maxSub` algorithm?  
(How long does it take to compute on an input of size  $n$ ?)

PRETTY BAD (we'll see how bad)

Q: Are there more efficient algorithms?

YES: We will see two of them

# Complexity of the Algorithm

Exercise: What is the complexity of the `maxSub` algorithm?  
(How long does it take to compute on an input of size  $n$ ?)

PRETTY BAD (we'll see how bad)

Q: Are there more efficient algorithms?

YES: We will see two of them

But First:

We see the Outline of the course

We review the basics of computational complexity



# Introduction and Prerequisites

---

# What is this course about?

COURSE CONTENTS:

# What is this course about?

## COURSE CONTENTS:

- **Advanced Data Structures**

How to store data efficiently

Graphs, Search Trees, Networks, Heaps

# What is this course about?

## COURSE CONTENTS:

- **Advanced Data Structures**

How to store data efficiently

Graphs, Search Trees, Networks, Heaps

- **Advanced Algorithms**

Efficient Algorithms to solve important problems

Fast search, route planning, network flow, scheduling

# What is this course about?

## COURSE CONTENTS:

- **Advanced Data Structures**

How to store data efficiently

Graphs, Search Trees, Networks, Heaps

- **Advanced Algorithms**

Efficient Algorithms to solve important problems

Fast search, route planning, network flow, scheduling

- **Programming Techniques**

Divide-and-Conquer, Dynamic Programming

# What is this course about?

## COURSE CONTENTS:

- **Advanced Data Structures**  
How to store data efficiently  
Graphs, Search Trees, Networks, Heaps
- **Advanced Algorithms**  
Efficient Algorithms to solve important problems  
Fast search, route planning, network flow, scheduling
- **Programming Techniques**  
Divide-and-Conquer, Dynamic Programming
- **Complexity Analysis**  
The Master Method, Amortized Complexity

Extra Subjects: Trendy Structures/Algorithms

- RSA Public-Key Cryptosystem
- Neural Networks (the Gradient-Descent Algorithm)
- Page Rank (the Google Search Algorithm)

# Prerequisites

- **Discrete Math**  
(Mathematics for Computer Scientists)  
IA Ch.3 and Appendices A and B
- **Basic Algorithms and Data Structures**  
(Algorithms Correctness and Efficiency)  
stacks, lists, trees (IA Ch.10)  
sorting (IA Ch.2)  
elements of computational complexity
- **Programming skills**  
In *some* programming language: C/C++, Java, Python, Haskell  
You need to understand Haskell code, but you don't have to write it



# Complexity Classes

---

# Running Times

We measure the complexity of an algorithm by the time it takes to execute:

$$T(n)$$

$n$  is the size of the input

$T(n)$  is the time it takes to run on inputs of size  $n$

# Running Times

We measure the **complexity of an algorithm** by the time it takes to execute:

$$T(n)$$

$n$  is the size of the input

$T(n)$  is the time it takes to run on inputs of size  $n$

It may take different times for different inputs of the same size  $n$ .

In general we mean the **worst-case** running time.

(Sometimes we're interested in the average running time.)

# Running Times

We measure the **complexity of an algorithm** by the time it takes to execute:

$$T(n)$$

$n$  is the size of the input

$T(n)$  is the time it takes to run on inputs of size  $n$

It may take different times for different inputs of the same size  $n$ .

In general we mean the **worst-case** running time.

(Sometimes we're interested in the average running time.)

We don't mean exact running time

(that depends on the implementation and machine)

but a measure of the number of **elementary computation steps**

# How to measure input and time

Strictly speaking we should measure:

- **input size** by the **number of bits** the input takes in memory
- **running time** by the number of **basic hardware operations** executed

# How to measure input and time

Strictly speaking we should measure:

- **input size** by the **number of bits** the input takes in memory
- **running time** by the number of **basic hardware operations** executed

In complexity theory we're usually more relaxed

- **input size** is often measured by **memory locations**, for example lists are measured by their lengths, trees by the number of nodes
- **running time** is measured by assuming that certain **elementary operations** (for example arithmetic, logical, pointer operations) take constant time (**which is usually false!**)

# How to measure input and time

Strictly speaking we should measure:

- **input size** by the **number of bits** the input takes in memory
- **running time** by the number of **basic hardware operations** executed

In complexity theory we're usually more relaxed

- **input size** is often measured by **memory locations**, for example lists are measured by their lengths, trees by the number of nodes
- **running time** is measured by assuming that certain **elementary operations** (for example arithmetic, logical, pointer operations) take constant time (**which is usually false!**)

Important is not the exact running time, but the **Complexity Class**: the algorithm runs in **linear**, or **quadratic**, ... or **exponential** time

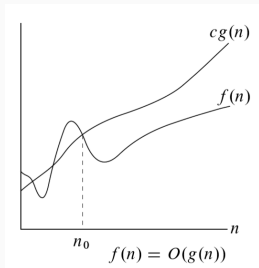
## Big- $O$ notation

The notation  $f(n) = O(g(n))$  intuitively means that the function  $f(n)$  grows at most as  $g(n)$



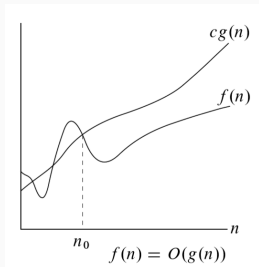
# Big- $O$ notation

The notation  $f(n) = O(g(n))$  intuitively means that the function  $f(n)$  grows at most as  $g(n)$



# Big- $O$ notation

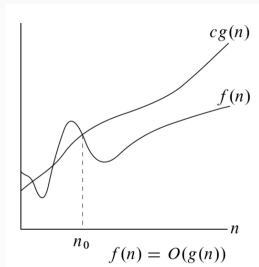
The notation  $f(n) = O(g(n))$  intuitively means that the function  $f(n)$  grows at most as  $g(n)$



- there is a multiple  $c$  of  $g(n)$  such that

# Big- $O$ notation

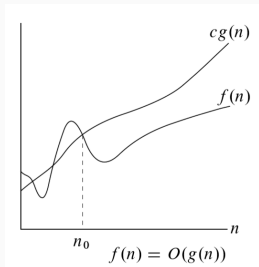
The notation  $f(n) = O(g(n))$  intuitively means that the function  $f(n)$  grows at most as  $g(n)$



- there is a multiple  $c$  of  $g(n)$  such that
- after a certain size  $n_0$

# Big- $O$ notation

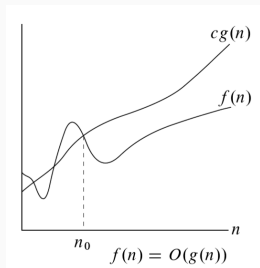
The notation  $f(n) = O(g(n))$  intuitively means that the function  $f(n)$  grows at most as  $g(n)$



- there is a multiple  $c$  of  $g(n)$  such that
- after a certain size  $n_0$
- $f(n)$  stays below  $cg(n)$

# Big- $O$ notation

The notation  $f(n) = O(g(n))$  intuitively means that the function  $f(n)$  grows at most as  $g(n)$

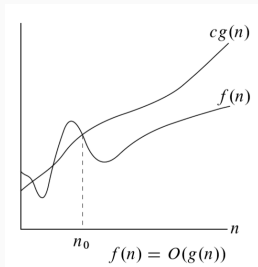


- there is a multiple  $c$  of  $g(n)$  such that
- after a certain size  $n_0$
- $f(n)$  stays below  $cg(n)$

$f(n) = O(g(n))$  means: there is a constant  $c$  and a number  $n_0$  such that for all  $n \geq n_0$ ,  $0 \leq f(n) \leq cg(n)$

# Big- $O$ notation

The notation  $f(n) = O(g(n))$  intuitively means that  
the function  $f(n)$  grows at most as  $g(n)$



- there is a multiple  $c$  of  $g(n)$  such that
- after a certain size  $n_0$
- $f(n)$  stays below  $cg(n)$

$f(n) = O(g(n))$  means: there is a constant  $c$  and a number  $n_0$  such that  
for all  $n \geq n_0$ ,  $0 \leq f(n) \leq cg(n)$

Formally we should write  $f \in O(g)$  where  $O(g)$  is the set of functions

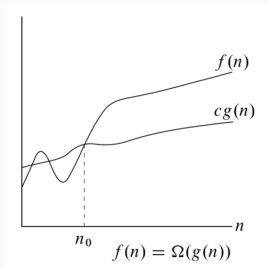
$$O(g) = \{f \mid \exists c, \exists n_0, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$

## Big- $\Omega$ notation

The notation  $f(n) = \Omega(g(n))$  intuitively means that the function  $f(n)$  grows at least as  $g(n)$

# Big- $\Omega$ notation

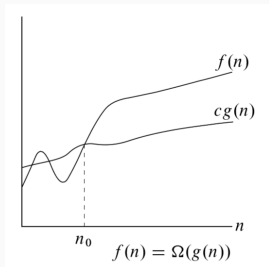
The notation  $f(n) = \Omega(g(n))$  intuitively means that the function  $f(n)$  grows at least as  $g(n)$





# Big- $\Omega$ notation

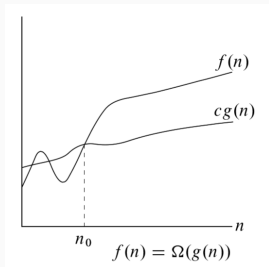
The notation  $f(n) = \Omega(g(n))$  intuitively means that the function  $f(n)$  grows at least as  $g(n)$



- there is a multiple  $c$  of  $g(n)$  such that

# Big- $\Omega$ notation

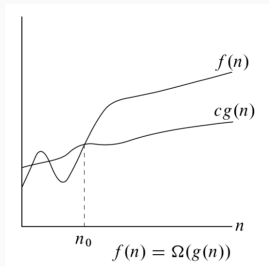
The notation  $f(n) = \Omega(g(n))$  intuitively means that the function  $f(n)$  grows at least as  $g(n)$



- there is a multiple  $c$  of  $g(n)$  such that
- after a certain size  $n_0$

# Big- $\Omega$ notation

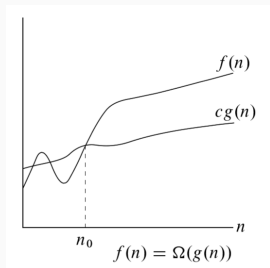
The notation  $f(n) = \Omega(g(n))$  intuitively means that the function  $f(n)$  grows at least as  $g(n)$



- there is a multiple  $c$  of  $g(n)$  such that
- after a certain size  $n_0$
- $f(n)$  stays above  $cg(n)$

# Big- $\Omega$ notation

The notation  $f(n) = \Omega(g(n))$  intuitively means that the function  $f(n)$  grows at least as  $g(n)$

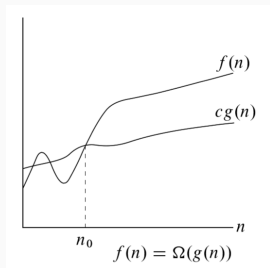


- there is a multiple  $c$  of  $g(n)$  such that
- after a certain size  $n_0$
- $f(n)$  stays above  $cg(n)$

$f(n) = \Omega(g(n))$  means: there is a constant  $c$  and a number  $n_0$  such that for all  $n \geq n_0$ ,  $0 \leq cg(n) \leq f(n)$

# Big-Ω notation

The notation  $f(n) = \Omega(g(n))$  intuitively means that the function  $f(n)$  grows at least as  $g(n)$



- there is a multiple  $c$  of  $g(n)$  such that
- after a certain size  $n_0$
- $f(n)$  stays above  $cg(n)$

$f(n) = \Omega(g(n))$  means: there is a constant  $c$  and a number  $n_0$  such that for all  $n \geq n_0$ ,  $0 \leq cg(n) \leq f(n)$

Formally we should write  $f \in \Omega(g)$  where  $\Omega(g)$  is the set of functions

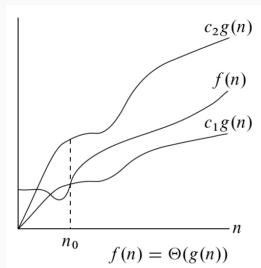
$$\Omega(g) = \{f \mid \exists c, \exists n_0, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$

## $\Theta$ notation

$\Theta(g(n))$  is the combination of  $O(g(n))$  and  $\Omega(g(n))$   
the function  $f(n)$  grows like  $g(n)$

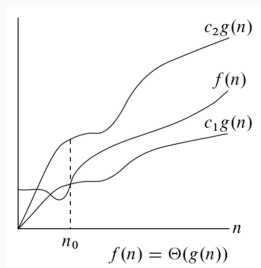
## $\Theta$ notation

$\Theta(g(n))$  is the combination of  $O(g(n))$  and  $\Omega(g(n))$   
the function  $f(n)$  grows like  $g(n)$



## $\Theta$ notation

$\Theta(g(n))$  is the combination of  $O(g(n))$  and  $\Omega(g(n))$   
the function  $f(n)$  grows like  $g(n)$

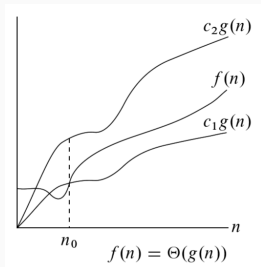


- there are two multiples  $c_1, c_2$  of  $g(n)$  such that



## $\Theta$ notation

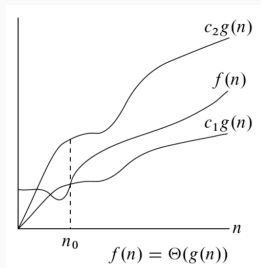
$\Theta(g(n))$  is the combination of  $O(g(n))$  and  $\Omega(g(n))$   
the function  $f(n)$  grows like  $g(n)$



- there are two multiples  $c_1, c_2$  of  $g(n)$  such that
- after a certain size  $n_0$

## $\Theta$ notation

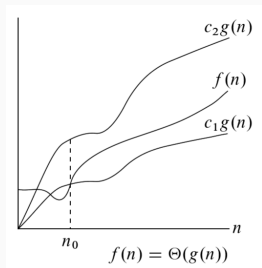
$\Theta(g(n))$  is the combination of  $O(g(n))$  and  $\Omega(g(n))$   
the function  $f(n)$  grows like  $g(n)$



- there are two multiples  $c_1, c_2$  of  $g(n)$  such that
- after a certain size  $n_0$
- $f(n)$  stays between  $c_1g(n)$  and  $c_2g(n)$

## Θ notation

$\Theta(g(n))$  is the combination of  $O(g(n))$  and  $\Omega(g(n))$   
the function  $f(n)$  grows like  $g(n)$

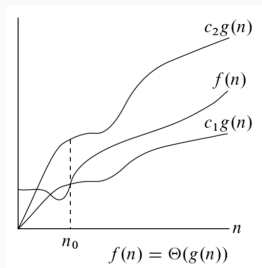


- there are two multiples  $c_1, c_2$  of  $g(n)$  such that
- after a certain size  $n_0$
- $f(n)$  stays between  $c_1g(n)$  and  $c_2g(n)$

$f(n) = \Theta(g(n))$  means: there are constants  $c_1, c_2$  and a number  $n_0$  such that  
for all  $n \geq n_0, 0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$

## Θ notation

$\Theta(g(n))$  is the combination of  $O(g(n))$  and  $\Omega(g(n))$   
the function  $f(n)$  grows like  $g(n)$



- there are two multiples  $c_1, c_2$  of  $g(n)$  such that
- after a certain size  $n_0$
- $f(n)$  stays between  $c_1g(n)$  and  $c_2g(n)$

$f(n) = \Theta(g(n))$  means: there are constants  $c_1, c_2$  and a number  $n_0$  such that  
for all  $n \geq n_0, 0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$

Formally we should write  $f \in \Theta(g)$  where

$$\Theta(g) = O(g) \cap \Omega(g)$$

There are also “small” versions of  $O$  and  $\Omega$  where the inequalities hold strictly

## $o$ and $\omega$ classes

There are also “small” versions of  $O$  and  $\Omega$   
where the inequalities hold strictly

$f(n) = o(g(n))$  means  $f(n)$  grows slower than  $g(n)$

There are also “small” versions of  $O$  and  $\Omega$   
where the inequalities hold strictly

$f(n) = o(g(n))$  means  $f(n)$  grows slower than  $g(n)$   
for every constant  $c > 0$  there exists a number  $n_0$   
such that for all  $n \geq n_0, 0 \leq f(n) < cg(n)$

$$o(g) = \{f \mid \forall c, \exists n_0, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$$

## $o$ and $\omega$ classes

There are also “small” versions of  $O$  and  $\Omega$   
where the inequalities hold strictly

$f(n) = o(g(n))$  means  $f(n)$  grows slower than  $g(n)$   
for every constant  $c > 0$  there exists a number  $n_0$   
such that for all  $n \geq n_0, 0 \leq f(n) < cg(n)$

$$o(g) = \{f \mid \forall c, \exists n_0, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$$

$f(n) = \omega(g(n))$  means  $f(n)$  grows faster than  $g(n)$



## $o$ and $\omega$ classes

There are also “small” versions of  $O$  and  $\Omega$   
where the inequalities hold strictly

$f(n) = o(g(n))$  means  $f(n)$  grows slower than  $g(n)$   
for every constant  $c > 0$  there exists a number  $n_0$   
such that for all  $n \geq n_0, 0 \leq f(n) < cg(n)$

$$o(g) = \{f \mid \forall c, \exists n_0, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$$

$f(n) = \omega(g(n))$  means  $f(n)$  grows faster than  $g(n)$   
for every constant  $c > 0$  there exists a number  $n_0$   
such that for all  $n \geq n_0, 0 \leq cg(n) < f(n)$

$$\omega(g) = \{f \mid \forall c, \exists n_0, \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$$

## Complexity of Maximum Subarray

---

# Analysis of the Naive Algorithm

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0 ≥ s then (0,j0,s0)
              else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

# Analysis of the Naive Algorithm

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0 ≥ s then (0,j0,s0)
              else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

When the input has length  $n = 1$ , it is a singleton  $[x]$

We immediately return the result  $(0, 0, x)$

**Constant time:** just write the output

$$T(1) = c_0$$

# Analysis of the Naive Algorithm

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0 ≥ s then (0,j0,s0)
              else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

For inputs with length  $n > 1$ , we must compute

- `(sums xs)` **linear time**: traverses the input list
- `argMax snd ...` **linear time**: traverses the sums
- `maxSub (tail xs)` **recursive call**

$$T(n) = c_1 n + T(n-1)$$

# Analysis of the Naive Algorithm

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0 ≥ s then (0,j0,s0)
              else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

$$T(1) = c_0$$

$$T(n) = c_1 n + T(n-1)$$

# Analysis of the Naive Algorithm

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0 ≥ s then (0,j0,s0)
              else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

$$T(1) = c_0$$

$$T(n) = c_1 n + T(n-1)$$

So the complexity is:

$$\begin{aligned} T(n) &= c_1 n + T(n-1) = c_1 n + c_1(n-1) + T(n-2) \\ &= c_1 n + c_1(n-1) + \cdots c_1 2 + T(1) \\ &= c_1 n + c_1(n-1) + \cdots c_1 2 + c_0 \\ &= c_1 \sum_{i=2}^{i=n} i + c_0 = c_1(n(n+1)/2 - 1) + c_0 \\ &= \Theta(n^2) \end{aligned}$$

# More Efficient Algorithms?

Exercise: Write a better algorithm for the Maximum Subarray Problem

Two ideas/strategies:



# More Efficient Algorithms?

Exercise: Write a better algorithm for the Maximum Subarray Problem

Two ideas/strategies:

1. A **divide-and-conquer** algorithm:
  - Split the list in two halves
  - Compute separately the maximum subarray of both halves
  - Compute the maximum *cross-over* subarray

This has complexity  $O(n \log n)$

# More Efficient Algorithms?

Exercise: Write a better algorithm for the Maximum Subarray Problem

Two ideas/strategies:

1. A **divide-and-conquer** algorithm:

- Split the list in two halves
- Compute separately the maximum subarray of both halves
- Compute the maximum *cross-over* subarray

This has complexity  $O(n \log n)$

2. A **linear time** algorithm (see Ex. 4.1-5 in IA)

- Traverse the list from left to right
- Keep track of the maximum subarray seen so far
- and the maximum subarray ending at the last seen element

This has complexity  $O(n)$