

### 3. Counting steps (Asymptotic analysis)

---

José Proença

Algorithms (CC4010) 2023/2024

CISTER – U.Porto, Porto, Portugal

<https://cister-labs.github.io/alg2324>



**CISTER** - Research Centre in  
Real-Time & Embedded  
Computing Systems

- Checking correctness of algorithms
- Measuring **precisely** performance of algorithms
- Measuring **asymptotically** performance of algorithms
- Analysing **recursive** functions
- Next: beyond worst-/best-case scenarios
  - **average time** of a single operation
  - analysis of sequences of operations (**amortised analysis**)

# Motivation

---

slides by Charles E. Leiserson  
pages 3-19



# Analysis of algorithms

*The theoretical study of computer-program performance and resource usage.*

What's more important than performance?

- modularity
- correctness
- maintainability
- functionality
- robustness
- user-friendliness
- programmer time
- simplicity
- extensibility
- reliability



# Why study algorithms and performance?

- Algorithms help us to understand *scalability*.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a *language* for talking about program behavior.
- Performance is the *currency* of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!



# The problem of sorting

**Input:** sequence  $\langle a_1, a_2, \dots, a_n \rangle$  of numbers.

**Output:** permutation  $\langle a'_1, a'_2, \dots, a'_n \rangle$  such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

**Example:**

**Input:** 8 2 4 9 3 6

**Output:** 2 3 4 6 8 9



# Insertion sort

“pseudocode”

```
INSERTION-SORT ( $A, n$ )    ▷  $A[1 \dots n]$ 
  for  $j \leftarrow 2$  to  $n$ 
    do  $key \leftarrow A[j]$ 
       $i \leftarrow j - 1$ 
      while  $i > 0$  and  $A[i] > key$ 
        do  $A[i+1] \leftarrow A[i]$ 
           $i \leftarrow i - 1$ 
       $A[i+1] = key$ 
```

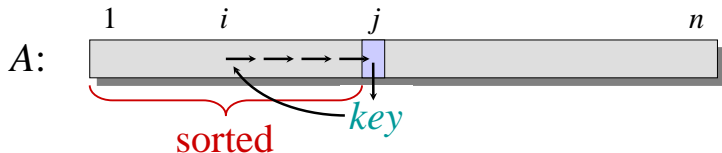




# Insertion sort

“pseudocode”

```
INSERTION-SORT ( $A, n$ )  $\triangleright A[1 \dots n]$   
  for  $j \leftarrow 2$  to  $n$   
    do  $key \leftarrow A[j]$   
       $i \leftarrow j - 1$   
      while  $i > 0$  and  $A[i] > key$   
        do  $A[i+1] \leftarrow A[i]$   
           $i \leftarrow i - 1$   
       $A[i+1] = key$ 
```





# Example of insertion sort

8      2      4      9      3      6



# Example of insertion sort





# Example of insertion sort





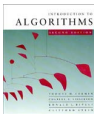
# Example of insertion sort



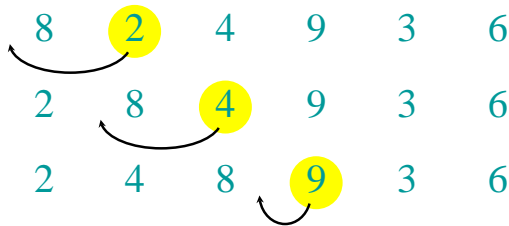


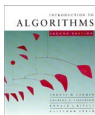
# Example of insertion sort





# Example of insertion sort





# Example of insertion sort





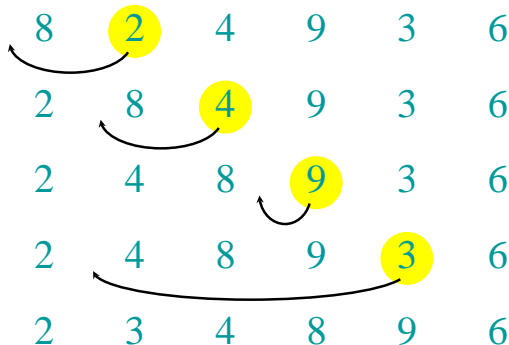


# Example of insertion sort



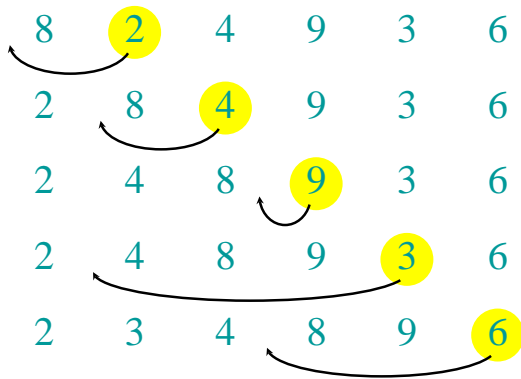


# Example of insertion sort





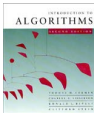
# Example of insertion sort





# Example of insertion sort





# Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

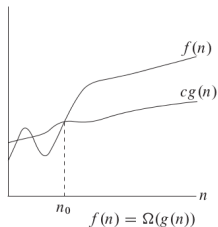
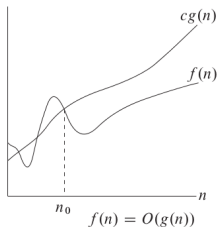
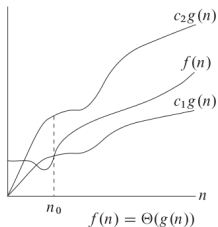
slides by Pedro Ribeiro, slides 2  
pages 1-2

# Asymptotic Analysis

Pedro Ribeiro

DCC/FCUP

2018/2019



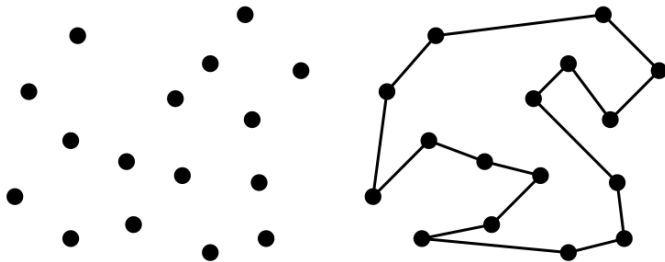
# Motivational Example - TSP

## Traveling Salesman Problem (Euclidean TSP version)

**Input:** a set  $S$  of  $n$  points in the plane

**Output:** the smallest possible path that starts on a point, visits all other points of  $S$  and then returns to the starting point.

An example:





slides by Pedro Ribeiro, slides 2  
pages 8-18

# Motivational Example - TSP

How to solve the problem then?

## A possible algorithm (exhaustive search a.k.a. "brute force")

$P_{min} \leftarrow$  any permutation of the points in  $S$

**For**  $P_i \leftarrow$  each of the permutations of points in  $S$

**If** ( $cost(P_i) < cost(P_{min})$ ) **then**

$P_{min} \leftarrow P_i$

**return** Path formed by  $P_{min}$

A correct algorithm, but **extremely slow!**

- $P(n) = n! = n \times (n - 1) \times \dots \times 1$
- For instance,  $P(20) = 2,432,902,008,176,640,000$
- For a set of 20 points, even the fastest computer in the world would not solve it! (how long would it take?)

# Motivational Example - TSP

- The present problem is a restricted version (euclidean) of one of the most well known "classic" hard problems, the **Travelling Salesman Problem (TSP)**
- This problem has **many possible applications**  
Ex: genomic analysis, industrial production, vehicle routing, ...
- There is no known **efficient solution** for this problem  
(with optimal results, not just approximated)
- The presented solution has  $\mathcal{O}(n!)$  complexity  
The Held-Karp algorithm has  $\mathcal{O}(2^n n^2)$  complexity  
(this notation will be the focus of this class)
- TSP belongs to the class of **NP-hard** problems  
The decision version belongs to the class of **NP-complete** problems  
(we will also talk about this at the end of the semester)

## An experience - how many instructions

- How many instructions per second on a current computer?  
(just an approximation, an order of magnitude)

On my notebook, about  $10^9$  instructions

- At this velocity, how much time for the following quantities of instructions?

Quant.	100	1000	10000
$N$	$< 0.01s$	$< 0.01s$	$< 0.01s$
$N^2$	$< 0.01s$	$< 0.01s$	$0.1s$
$N^3$	$< 0.01s$	$1.00s$	$16 \text{ min}$
$N^4$	$0.1s$	$16 \text{ min}$	$115 \text{ days}$
$2^N$	$10^{13} \text{ years}$	$10^{284} \text{ years}$	$10^{2993} \text{ years}$
$n!$	$10^{141} \text{ years}$	$10^{2551} \text{ years}$	$10^{35642} \text{ years}$

## An experience: - Permutations

- Let's go back to the idea of **permutations**

### Exemple: the 6 permutations of $\{1, 2, 3\}$

1 2 3

1 3 2

2 1 3

2 3 1

3 1 2

3 2 1

- Recall that the number of permutations can be computed as:

$$P(n) = n! = n \times (n - 1) \times \dots \times 1$$

(do you understand the intuition on the formula?)

## An experience: - Permutations

- What is the execution time of a program that goes through all permutations?

(the following times are approximated, on my notebook)

(what I want to show is **order of growth**)

**n** ≤ 7: < 0.001s

**n** = 8: 0.001s

**n** = 9: 0.016s

**n** = 10: 0.185s

**n** = 11: 2.204s

**n** = 12: 28.460s

...

**n** = 20: 5000 years !

How many permutations per second?

About  $10^7$

# On computer speed

- Will a **faster computer** be of any help? **No!**  
If  $n = 20 \rightarrow 5000$  years, hypothetically:
  - ▶ 10x faster would still take 500 years
  - ▶ 5,000x would still take 1 year
  - ▶ 1,000,000x faster would still take two days, but  
 $n = 21$  would take more than a month  
 $n = 22$  would take more than a year!
- The **growth rate** of the execution time is what matters!

## Algorithmic performance vs Computer speed

A better algorithm on a slower computer **will always win** against a worst algorithm on a faster computer, for sufficiently large instances

# Why worry?

- What can we do with execution time/memory analysis?

## Prediction

How much time/space does an algorithm need to solve a problem? How does it scale? Can we provide guarantees on its running time/memory?

## Comparison

Is an algorithm  $A$  better than an algorithm  $B$ ? Fundamentally, what is the best we can possibly do on a certain problem?

- We will study a **methodology** to answer these questions
- We will focus mainly on execution time analysis



# Random Access Machine (RAM)

- We need a **model** that is **generic** and **independent** from the language and the machine.
- We will consider a Random Access Machine (**RAM**)
  - ▶ Each **simple operation** (ex:  $+$ ,  $-$ ,  $\leftarrow$ , **If**) takes **1 step**
  - ▶ Loops and procedures, for example, are not simple instructions!
  - ▶ Each **access to memory** takes also **1 step**
- We can measure execution time by... **counting the number of steps as a function of the input size  $n$ :  $T(n)$ .**
- Operations are **simplified**, but this is useful  
Ex: summing two integers does not cost the same as dividing two reals, but we will see that on a global vision, these specific values are not important

# Random Access Machine (RAM)

## A counting example

### A simple program

```
int count = 0;
for (int i=0; i<n; i++)
    if (v[i] == 0) count++
```

Let's count the number of simple operations:

Variable declarations	2
Assignments:	2
"Less than" comparisons	$n + 1$
"Equality" comparisons:	$n$
Array access	$n$
Increment	between $n$ and $2n$

# Random Access Machine (RAM)

## A counting example

### A simple program

```
int count = 0;
for (int i=0; i<n; i++)
    if (v[i] == 0) count++
```

Total number of steps on the **worst** case:

$$T(n) = 2 + 2 + (n + 1) + n + n + 2n = 5 + 5n$$

Total number of steps on the **best** case:

$$T(n) = 2 + 2 + (n + 1) + n + n + n = 5 + 4n$$

# Types of algorithm analysis

**Worst Case** analysis: (the most common)

- $T(n)$  = maximum amount of time for any input of size  $n$

**Average Case** analysis: (sometimes)

- $T(n)$  = average time on all inputs of size  $n$
- Implies knowing the statistical distribution of the inputs

**Best Case** analysis: ("deceiving")

- It's almost like "cheating" with an algorithm that is fast just for **some** of the inputs

1. Precise analysis: counting operations
2. Approximate analysis – Asymptotic notation( $O, \Theta, \Omega, o, \omega$ )

# Counting operations

---

# Simpler counting

```
int count = 0;
for (int i=0; i<n; i++)
    if (v[i] == 0) count++
```

## RAM

- worst-case:  $T(n) = 5 + 5n$
- best-case:  $T(n) = 5 + 4n$

## #array-accesses + #count-increments

- worst-case:  $T(n) = 2n$
- best-case:  $T(n) = n$
- average-case:

$$\overline{T}(n) = n + \sum_{0 \leq r < n} P(v[r] = 0)$$

# Exercises

```
void bubbleSort(int v[], int N){
    int i, j;
    for (i=N-1; i>0; i--)
        for (j=0; j<i; j++)
            if (v[j] > v[j+1])
                swap(v,j,j+1);
}
```

```
void iSort(int v[], int N){
    int i, j;
    for (i=1; i<N; i++)
        for (j=i; j>0 && v[j-1]>v[j];
            j--)
            swap(v,j,j-1);
}
```

**Ex. 3.1:** What is the best and worst case wrt comparisons between array elements?

**Ex. 3.2:** What is the best and worst case wrt swaps?

**Ex. 3.3:** How many of these operations are performed in both cases?



# Exercises

```
int mult1 (int x, int y){
    int a, b, r;
    a=x; b=y; r=0;
    while (a!=0){
        r = r+b;
        a = a-1;
    }
    return r;
}
```

```
int mult2 (int x, int y){
    int a, b, r;
    a=x; b=y; r=0;
    while (a!=0) {
        if (a%2 == 1) r = r+b;
        a=a/2;
        b=b*2;
    }
    return r;
}
```

**Ex. 3.4:** In each case, how many primitive operations (+ - \*2 /2 %2) are performed?

Note: In mult2, consider the size  $N$  as the number of bits used to represent  $x$  and  $y$ ; e.g., with 5 bits you can represent a positive integer until 31.

# Exercises

```
int maxgrow(int v[], int N) {  
    int r = 1, i = 0, m;  
    while (i < N-1) {  
        m = grow(v+i, N-i);  
        if (m > r) r = m;  
        i++;  
    }  
    return r;  
}
```

```
int grow(int v[], int N) {  
    int i;  
    for (i=1; i < N; i++)  
        if (v[i] < v[i-1]) break;  
    return i;  
}
```

**Ex. 3.5:** What is the best and worst case for maxgrow wrt comparisons of array elements?

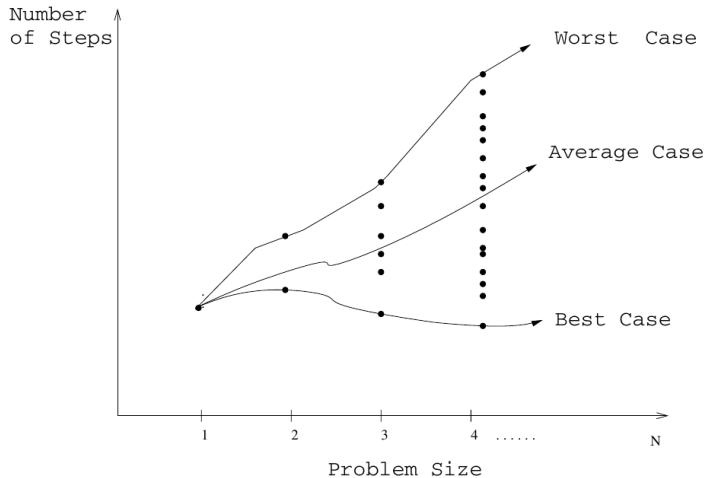
**Ex. 3.6:** How many comparisons are in each case?

# Asymptotic Notation

---

slides by Pedro Ribeiro, slides 2  
pages 19-23

# Types of algorithm analysis



# Asymptotic Notation

We need a mathematical tool to **compare functions**

On algorithm analysis we use **Asymptotic Analysis**:

- "Mathematically": studying the behaviour of **limits** (as  $n \rightarrow \infty$ )
- Computer Science: studying the behaviour for arbitrary large input  
or  
"describing" **growth rate**
- A very specific **notation** is used:  $O, \Omega, \Theta, o, \omega$
- It allows to focus on **orders of growth**

# Asymptotic Notation

## Definitions

$$f(n) = \mathcal{O}(g(n))$$

It means that  $c \times g(n)$  is an **upper bound** of  $f(n)$

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

It means that  $c \times g(n)$  is a **lower bound** of  $f(n)$

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

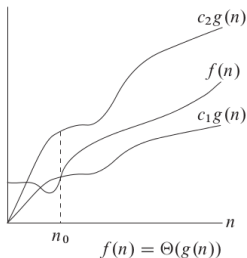
It means that  $c_1 \times g(n)$  is a **lower bound** of  $f(n)$  and  $c_2 \times g(n)$  is an **upper bound** of  $f(n)$

Note:  $\in$  could be used instead of  $=$

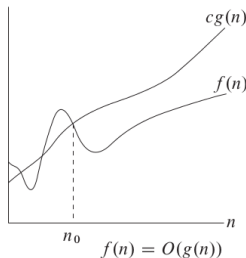
# Asymptotic Notation

A graphical depiction

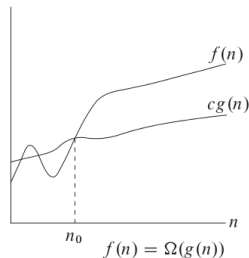
$\Theta$



$O$



$\Omega$



The definitions imply an  $n$  from which the function is bounded. The small values of  $n$  do not "matter".



# Asymptotic Notation

## Formalization

- $\mathbf{f(n) = \mathcal{O}(g(n))}$  if there exist positive constants  $n_0$  and  $c$  such that  $f(n) \leq c \times g(n)$  for all  $n \geq n_0$
- $\mathbf{f(n) = \Omega(g(n))}$  if there exist positive constants  $n_0$  and  $c$  such that  $f(n) \geq c \times g(n)$  for all  $n \geq n_0$
- $\mathbf{f(n) = \Theta(g(n))}$  if there exist positive constants  $n_0$ ,  $c_1$  and  $c_2$  such that  $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$  for all  $n \geq n_0$
- $\mathbf{f(n) = o(g(n))}$  if for any positive constant  $c$  there exists  $n_0$  such that  $f(n) < c \times g(n)$  for all  $n \geq n_0$
- $\mathbf{f(n) = \omega(g(n))}$  if for any positive constant  $c$  there exists  $n_0$  such that  $f(n) > c \times g(n)$  for all  $n \geq n_0$

# Examples

## Big Oh ( $\mathcal{O}$ )

$$3n^2 - 100n + 6 \stackrel{?}{=} \mathcal{O}(n^2)$$

$$3n^2 - 100n + 6 \stackrel{?}{=} \mathcal{O}(n^3)$$

$$3n^2 - 100n + 6 \stackrel{?}{=} \mathcal{O}(n)$$

## Big Omega ( $\Omega$ )

$$3n^2 - 100n + 6 \stackrel{?}{=} \Omega(n^2)$$

$$3n^2 - 100n + 6 \stackrel{?}{=} \Omega(n^3)$$

$$3n^2 - 100n + 6 \stackrel{?}{=} \Omega(n)$$

## Big Theta ( $\Theta$ )

$$3n^2 - 100n + 6 \stackrel{?}{=} \Theta(n^2)$$

$$3n^2 - 100n + 6 \stackrel{?}{=} \Theta(n^3)$$

$$3n^2 - 100n + 6 \stackrel{?}{=} \Theta(n)$$

# Examples

## Big Oh ( $\mathcal{O}$ )

$$3n^2 - 100n + 6 = \mathcal{O}(n^2) \quad \text{because } 3n^2 > 3n^2 - 100n + 6$$

$$3n^2 - 100n + 6 = \mathcal{O}(n^3) \quad \text{because } 0.01n^3 > 3n^2 - 100n + 6$$

$$3n^2 - 100n + 6 \neq \mathcal{O}(n) \quad \text{because } c \cdot n < 3n^2 \text{ when } n > c$$

## Big Omega ( $\Omega$ )

$$3n^2 - 100n + 6 =? \Omega(n^2)$$

$$3n^2 - 100n + 6 =? \Omega(n^3)$$

$$3n^2 - 100n + 6 =? \Omega(n)$$

## Big Theta ( $\Theta$ )

$$3n^2 - 100n + 6 =? \Theta(n^2)$$

$$3n^2 - 100n + 6 =? \Theta(n^3)$$

$$3n^2 - 100n + 6 =? \Theta(n)$$

# Examples

## Big Oh ( $\mathcal{O}$ )

$$3n^2 - 100n + 6 = \mathcal{O}(n^2) \quad \text{because} \quad 3n^2 > 3n^2 - 100n + 6$$

$$3n^2 - 100n + 6 = \mathcal{O}(n^3) \quad \text{because} \quad 0.01n^3 > 3n^2 - 100n + 6$$

$$3n^2 - 100n + 6 \neq \mathcal{O}(n) \quad \text{because} \quad c \cdot n < 3n^2 \text{ when } n > c$$

## Big Omega ( $\Omega$ )

$$3n^2 - 100n + 6 = \Omega(n^2) \quad \text{because} \quad 2.99n^2 < 3n^2 - 100n + 6$$

$$3n^2 - 100n + 6 \neq \Omega(n^3) \quad \text{because} \quad n^3 > 3n^2 - 100n + 6$$

$$3n^2 - 100n + 6 = \Omega(n) \quad \text{because} \quad 10^{10^{10}} n < 3n^2 - 100 + 6$$

## Big Theta ( $\Theta$ )

$$3n^2 - 100n + 6 \stackrel{?}{=} \Theta(n^2)$$

$$3n^2 - 100n + 6 \stackrel{?}{=} \Theta(n^3)$$

$$3n^2 - 100n + 6 \stackrel{?}{=} \Theta(n)$$

# Examples

## Big Oh ( $\mathcal{O}$ )

$$3n^2 - 100n + 6 = \mathcal{O}(n^2) \quad \text{because} \quad 3n^2 > 3n^2 - 100n + 6$$

$$3n^2 - 100n + 6 = \mathcal{O}(n^3) \quad \text{because} \quad 0.01n^3 > 3n^2 - 100n + 6$$

$$3n^2 - 100n + 6 \neq \mathcal{O}(n) \quad \text{because} \quad c \cdot n < 3n^2 \text{ when } n > c$$

## Big Omega ( $\Omega$ )

$$3n^2 - 100n + 6 = \Omega(n^2) \quad \text{because} \quad 2.99n^2 < 3n^2 - 100n + 6$$

$$3n^2 - 100n + 6 \neq \Omega(n^3) \quad \text{because} \quad n^3 > 3n^2 - 100n + 6$$

$$3n^2 - 100n + 6 = \Omega(n) \quad \text{because} \quad 10^{10^{10}} n < 3n^2 - 100 + 6$$

## Big Theta ( $\Theta$ )

$$3n^2 - 100n + 6 = \Theta(n^2) \quad \text{because} \quad \mathcal{O} \text{ and } \Omega$$

$$3n^2 - 100n + 6 \neq \Theta(n^3) \quad \text{because} \quad \mathcal{O} \text{ only}$$

$$3n^2 - 100n + 6 \neq \Theta(n) \quad \text{because} \quad \Omega \text{ only}$$

slides by Pedro Ribeiro, slides 2  
pages 24-31

# Asymptotic Notation

## Analogy

Comparison between two functions  $f$  and  $g$  and two numbers  $a$  and  $b$ :

$f(n) = \mathcal{O}(g(n))$	is like	$a \leq b$	upper bound	at least as good as
$f(n) = \Omega(g(n))$	is like	$a \geq b$	lower bound	at least as bad as
$f(n) = \Theta(g(n))$	is like	$a = b$	tight bound	as good as
$f(n) = \mathbf{o}(g(n))$	is like	$a < b$	strict upper b.	strictly better than
$f(n) = \omega(g(n))$	is like	$a > b$	strict lower b.	strictly worst than

# Asymptotic Notation

## A few consequences

- $f(n) = \Theta(g(n)) \rightarrow f(n) = \mathcal{O}(g(n))$  and  $f(n) = \Omega(g(n))$
  - $f(n) = \mathcal{O}(g(n)) \rightarrow f(n) \neq \omega(g(n))$
  - $f(n) = \Omega(g(n)) \rightarrow f(n) \neq \mathfrak{o}(g(n))$
  - $f(n) = \mathfrak{o}(g(n)) \rightarrow f(n) \neq \Omega(g(n))$
  - $f(n) = \omega(g(n)) \rightarrow f(n) \neq \mathcal{O}(g(n))$
- 
- $f(n) = \Theta(g(n)) \rightarrow g(n) = \Theta(f(n))$
  - $f(n) = \mathcal{O}(g(n)) \rightarrow g(n) = \Omega(f(n))$
  - $f(n) = \Omega(g(n)) \rightarrow g(n) = \mathcal{O}(f(n))$
  - $f(n) = \mathfrak{o}(g(n)) \rightarrow g(n) = \omega(f(n))$
  - $f(n) = \omega(g(n)) \rightarrow g(n) = \mathfrak{o}(f(n))$



# Asymptotic Notation

## A few practical rules

- **Multiplying by a constant** does not affect:

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

$$99 \times n^2 = \Theta(n^2)$$

- On a polynomial of the form  $a_x n^x + a_{x-1} n^{x-1} + \dots + a_2 n^2 + a_1 n + a_0$  we can focus on the term with the **largest exponent**:

$$3n^3 - 5n^2 + 100 = \Theta(n^3)$$

$$6n^4 - 20^2 = \Theta(n^4)$$

$$0.8n + 224 = \Theta(n)$$

- More than that, on a sum we can focus on the **dominant** term:

$$2^n + 6n^3 = \Theta(2^n)$$

$$n! - 3n^2 = \Theta(n!)$$

$$n \log n + 3n^2 = \Theta(n^2)$$

# Asymptotic Notation

## Dominance

When is a function **better** than another?

- If we want to minimize time, "**smaller**" functions are **better**
- A function **dominates** another if as  $n$  grows it keeps getting larger
- Mathematically:  $f(n) \gg g(n)$  if  $\lim_{n \rightarrow \infty} g(n)/f(n) = 0$

### Dominance Relations

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$$

# Asymptotic Growth

## A practical view

If an operation takes  $10^{-9}$  seconds...

	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$	$n!$
10	< 0.01s	< 0.01s	< 0.01s	< 0.01s	< 0.01s	< 0.01s	< 0.01s
20	< 0.01s	< 0.01s	< 0.01s	< 0.01s	< 0.01s	< 0.01s	77 years
30	< 0.01s	< 0.01s	< 0.01s	< 0.01s	< 0.01s	1.07s	
40	< 0.01s	< 0.01s	< 0.01s	< 0.01s	< 0.01s	18.3 min	
50	< 0.01s	< 0.01s	< 0.01s	< 0.01s	< 0.01s	13 days	
100	< 0.01s	< 0.01s	< 0.01s	< 0.01s	< 0.01s	$10^{13}$ years	
$10^3$	< 0.01s	< 0.01s	< 0.01s	< 0.01s	1s		
$10^4$	< 0.01s	< 0.01s	< 0.01s	0.1s	16.7 min		
$10^5$	< 0.01s	< 0.01s	< 0.01s	10s	11 days		
$10^6$	< 0.01s	< 0.01s	0.02s	16.7 min	31 years		
$10^7$	< 0.01s	0.01s	0.23s	1.16 days			
$10^8$	< 0.01s	0.1s	2.66s	115 days			
$10^9$	< 0.01s	1s	29.9s	31 years			

# Asymptotic Notation

## Common Functions

Function	Name	Examples
1	constant	summing two numbers
$\log n$	logarithmic	binary search, inserting in a heap
$n$	linear	1 loop to find maximum value
$n \log n$	linearithmic	sorting (ex: mergesort, heapsort)
$n^2$	quadratic	2 loops (ex: verifying, bubblesort)
$n^3$	cubic	3 loops (ex: Floyd-Warshall)
$2^n$	exponential	exhaustive search (ex: subsets)
$n!$	factorial	all permutations

# Asymptotic Growth

## Drawing functions

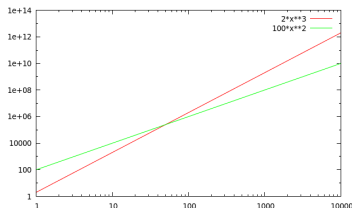
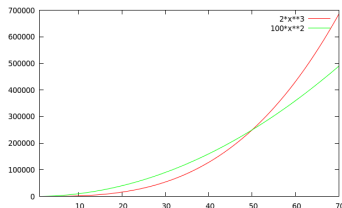
An useful program for drawing functions is **gnuplot**.

(comparing  $2n^3$  with  $100n^2$  for  $1 \leq n \leq 100$ )

```
gnuplot> plot [1:70] 2*x**3, 100*x**2
```

```
gnuplot> set logscale xy 10
```

```
gnuplot> plot [1:10000] 2*x**3, 100*x**2
```



(which grows faster:  $\sqrt{n}$  or  $\log_2 n$ ?)

```
gnuplot> plot [1:10000000] sqrt(x), log(x)/log(2)
```

# Asymptotic Analysis

## A few more examples

- A program has two pieces of code  $A$  and  $B$ , executed one after the other, with  $A$  running in  $\Theta(n \log n)$  and  $B$  in  $\Theta(n^2)$ .  
The program runs in  $\Theta(n^2)$ , because  $n^2 \gg n \log n$
- A program calls  $n$  times a function  $\Theta(\log n)$ , and then it calls again  $n$  times another function  $\Theta(\log n)$   
The program runs in  $\Theta(n \log n)$
- A program has 5 loops, all called sequentially, each one of them running in  $\Theta(n)$   
The program runs in  $\Theta(n)$
- A program  $P_1$  has execution time proportional to  $100 \times n \log n$ . Another program  $P_2$  runs in  $2 \times n^2$ .  
Which one is more efficient?  
 $P_1$  is more efficient because  $n^2 \gg n \log n$ . However, for a small  $n$ ,  $P_2$  is quicker and it might make sense to have a program that calls  $P_1$  or  $P_2$  depending on  $n$ .

slides by Pedro Ribeiro, exercises 2  
pages 1-2

## Exercises #2

### Asymptotic Analysis

#### Theoretical Background

Remember the asymptotic notation:

- $\mathbf{f(n) = O(g(n))}$  if there exist positive constants  $n_0$  and  $c$  such that  $f(n) \leq cg(n)$  for all  $n \geq n_0$ .
- $\mathbf{f(n) = \Omega(g(n))}$  if there exist positive constants  $n_0$  and  $c$  such that  $f(n) \geq cg(n)$  for all  $n \geq n_0$ .
- $\mathbf{f(n) = \Theta(g(n))}$  if there exist positive constants  $n_0$ ,  $c_1$  and  $c_2$  such that  $c_1g(n) \leq f(n) \leq c_2g(n)$  for all  $n \geq n_0$ .
- $\mathbf{f(n) = o(g(n))}$  if for any positive constant  $c$  there exists  $n_0$  such that  $f(n) < cg(n)$  for all  $n \geq n_0$ .
- $\mathbf{f(n) = \omega(g(n))}$  if for any positive constant  $c$  there exists  $n_0$  such that  $f(n) > cg(n)$  for all  $n \geq n_0$ .

#### Asymptotic Notation

1. Is  $2^{n+1} = O(2^n)$ ? Is  $2^{2n} = O(2^n)$ . Justify your answer with brief proofs.
2. For each pair of functions  $f(n)$  and  $g(n)$ , indicate whether  $f(n)$  is  $O$ ,  $o$ ,  $\Omega$ ,  $\omega$ , or  $\Theta$  of  $g(n)$ . Your answer should be in the form of a "yes" or "no" for each cell of the table.

	$f(n)$	$g(n)$	$O$	$o$	$\Omega$	$\omega$	$\Theta$
(a)	$2n^2 - 10n^2$	$25n^2 + 37n$					
(b)	56	$\log_2 30$					
(c)	$\log_3 n$	$\log_2 n$					
(d)	$n^3$	$3^n$					
(e)	$n!$	$2^n$					
(f)	$n!$	$n^n$					
(g)	$n \log_2 n + n^2$	$n^2$					
(h)	$\sqrt{n}$	$\log_2 n$					
(i)	$\log_3(\log_3 n)$	$\log_3 n$					
(j)	$\log_2 n$	$\log_2 n^2$					



3. For each of the following conjectures, indicate if they are true or false, explaining why.

You can assume that functions  $f(n)$  and  $g(n)$  are asymptotically positive, i.e., they are positive from some point on ( $\exists n_0 : f(n) > 0$  for all  $n \geq n_0$ )

- (a)  $f(n) = O(g(n))$  implies that  $g(n) = O(f(n))$
- (b)  $f(n) = O(g(n))$  implies that  $g(n) = \Omega(f(n))$
- (c)  $f(n) + g(n) = \Theta(\min(f(n), g(n)))$
- (d)  $f(n) + g(n) = \Theta(\max(f(n), g(n)))$
- (e)  $(n + c)^k = \Theta(n^k)$ , where  $c$  and  $k$  are positive integer constants
- (f)  $f(n) + o(f(n)) = \Theta(f(n))$
- (g)  $n^2 = \Theta(16^{\log_4 n})$

---

### Growth Ratio

4. Imagine a program  $A$  running with time complexity  $\Theta(f(n))$ , taking  $t$  seconds for an input of size  $k$ . What would your estimation be for the execution time for an input of size  $2k$  for the following functions:  $n$ ,  $n^2$ ,  $n^3$ ,  $2^n$ ,  $\log_2 n$ . Is this growth ratio constant for any  $k$  or is it changing?
5. Consider two programs implementing algorithms  $A$  and  $B$ , both trying to solve the same problem for an input of size  $n$ . They measured the execution times for test cases of different sizes and got the following table:

Algorithm	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$
$A$	0.003s	0.024s	0.081s	0.192s	0.375s
$B$	0.040s	0.160s	0.360s	0.640s	1.000s

- (a) Which program is more efficient? Why?
- (b) Could you produce a program that uses both algorithms in order to produce an algorithm  $C$  that would be at least as good as  $A$  and  $B$  for any test case?

# Analysis of recursive functions

---

# Binary search

To analyse the complexity of a recursive function, we typically define the time  $T$  using *recurring equations*.

```
int bsearch(int x, int v[], int N){
    int i;
    if (N<=0) i = -1;
    else {
        m = N/2;
        if (v[m]==x) i = m;
        else if (v[m] > x)
            i = bsearch(x, v, m);
        else {
            i = bsearch(x, v+m+1, N-m-1);
            if (i!=-1) i = i+m+1
        }
    }
    return i ;
}
```

Counting the number of comparisons with array elements:

$$T(N) = \begin{cases} 0 & \text{if } N = 0 \\ T(N/2) + 2 & \text{if } N > 0 \end{cases}$$

# Binary search

To analyse the complexity of a recursive function, we typically define the time  $T$  using *recurring equations*.

```
int bsearch(int x, int v[], int N){
    int i;
    if (N<=0) i = -1;
    else {
        m = N/2;
        if (v[m]==x) i = m;
        else if (v[m] > x)
            i = bsearch(x, v, m);
        else {
            i = bsearch(x, v+m+1, N-m-1);
            if (i!=-1) i = i+m+1
        }
    }
    return i ;
}
```

$$(T(N) = T(N/2)+2 \text{ if } N>0)$$

$$T(0) = 0$$

$$T(1) = T(2^0) = 2$$

$$T(2) = T(2^1) = 2 + T(2/2) = 2 + 2 = 4$$

$$T(4) = T(2^2) = 2 + T(4/2) = 2 + 2 + 2 = 6$$

...

$$T(2^i) = \underbrace{2 + 2 + \cdots + 2}_{i\text{-times}} + 2 = 2i + 2$$

# Binary search

To analyse the complexity of a recursive function, we typically define the time  $T$  using *recurring equations*.

```
int bsearch(int x, int v[], int N){
    int i;
    if (N<=0) i = -1;
    else {
        m = N/2;
        if (v[m]==x) i = m;
        else if (v[m] > x)
            i = bsearch(x, v, m);
        else {
            i = bsearch(x, v+m+1, N-m-1);
            if (i!=-1) i = i+m+1
        }
    }
    return i ;
}
```

$$(T(N) = T(N/2)+2 \text{ if } N>0)$$

$$T(0) = 0$$

$$T(1) = T(2^0) = 2$$

$$T(2) = T(2^1) = 2 + T(2/2) = 2 + 2 = 4$$

$$T(4) = T(2^2) = 2 + T(4/2) = 2 + 2 + 2 = 6$$

...

$$T(2^i) = \underbrace{2 + 2 + \dots + 2}_{i\text{-times}} + 2 = 2i + 2$$

$$\begin{aligned} T(N) &= T(2^{\log_2(N)}) \\ &= 2 * \log_2(N) + 2 \quad (= \Theta(\log(N))) \end{aligned}$$

slides by Pedro Ribeiro, slides 2  
pages 38-61

# Divide and Conquer

We are often interested in algorithms that are expressed in a **recursive** way

Many of these algorithms follow the **divide and conquer** strategy:

## Divide and Conquer

**Divide** the problem in a set of subproblems which are smaller instances of the same problem

**Conquer** the subproblems solving them recursively. If the problem is small enough, solve it directly.

**Combine** the solutions of the smaller subproblems on a solution for the original problem

# Divide and Conquer

## MergeSort

We now describe the **MergeSort** algorithm for sorting an array of size  $n$

### **MergeSort**

**Divide:** partition the initial array in two halves

**Conquer:** recursively sort each half. If we only have one number, it is sorted.

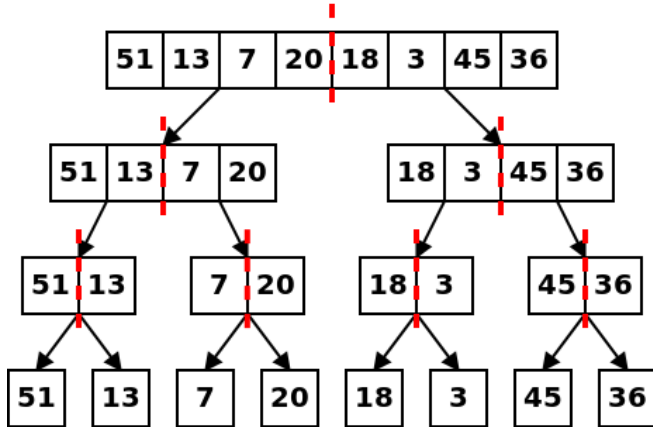
**Combine:** merge the two sorted halves in a final sorted array



# Divide and Conquer

## MergeSort

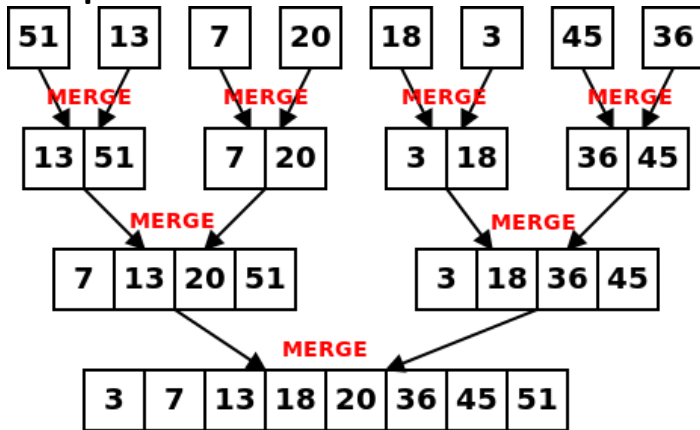
Divide:



# Divide and Conquer

## MergeSort

Conquer:



# Divide and Conquer

## MergeSort

What is the **execution time** of this algorithm?

- **D(n)** - Time to partition an array of size  $n$  in two halves
- **M(n)** - Time to merge two sorted arrays of size  $n$
- **T(n)** - Time for a MergeSort on an array of size  $n$

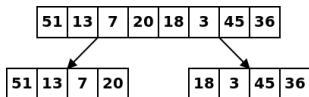
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ D(n) + 2T(n/2) + M(n) & \text{if } n > 1 \end{cases}$$

In practice, we are ignoring certain details, but it suffices  
(ex: when  $n$  is odd, the size of subproblem is not exactly  $n/2$ )

# Divide and Conquer

## MergeSort

$D(n)$  - Time to partition an array of size  $n$  in two halves



We can do it in constant time!  $\Theta(1)$

`mergesort(a,b)`: (sort from position  $a$  to  $b$ )

In the beginning, call `mergesort(0,n-1)`

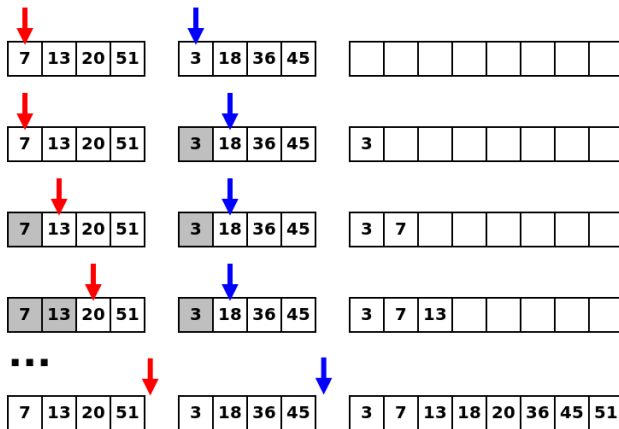
Let  $m = \lfloor (a + b)/2 \rfloor$  (middle position)

Call `mergesort(a,m)` and `mergesort(m+1,b)`

# Divide and Conquer

## MergeSort

$M(n)$  - Time to merge two sorted arrays of size  $n$



We can do it in linear time!  $\Theta(n)$  ( $2n$  comparisons)

# Divide and Conquer

## MergeSort

Back to the mergesort recurrence:

- **D(n)** - Time to partition an array of size  $n$  in two halves
- **M(n)** - Time to merge two sorted arrays of size  $n$
- **T(n)** - Time for a MergeSort on an array of size  $n$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ D(n) + 2T(n/2) + M(n) & \text{if } n > 1 \end{cases}$$

becomes

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

For sufficiently small inputs, an algorithm generally takes constant time. This means that for a small  $n$ , we have  $T(n) = \Theta(1)$

For convenience, we can generally **omit the boundary condition of the recurrence**.

Examples:

- Mergesort:  $T(n) = 2T(n/2) + \Theta(n)$
- Binary Search:  $T(n) = T(n/2) + \Theta(1)$
- Finding Maximum with tail recursion:  $T(n) = T(n-1) + \Theta(1)$

How to **solve** recurrences like this?

# Recurrences

## Solving

We are going to talk about 4 methods:

- **Unrolling:** unroll the recurrence to obtain an expression (ex: summation) you can work with
- **Substitution:** guess the answer and prove by induction
- **Recursion Tree:** draw a tree representing the recursion and sum all the work done in the nodes
- **Master Theorem:** If the recurrence is of the form  $aT(n/b) + cn^k$ , the answer follows a certain pattern



# Solving Recurrences

## Unrolling Method

Some recurrences can be solved by **unrolling** them to get a summation:

$$T(n) = T(n-1) + \Theta(n) = \Theta(n) + \Theta(n-1) + \Theta(n-2) + \dots + \Theta(1)$$

$$T(n) = T(n-1) + cn = cn + c(n-1) + c(n-2) + \dots + c$$

There are  $n$  terms and each one is at most  $cn$ , so the summation is **at most**  $cn^2$ .

Similarly, since the first  $n/2$  terms are each **at least**  $cn/2$ , this summation is at least  $(n/2)(cn/2) = cn^2/4$ .

Given this, the recurrence is  $\Theta(n^2)$ .

We could have also used arithmetic progressions:

$$T(n) = c[n + (n-1) + \dots + c] = c \frac{(n+c)n}{2} = cn^2 + c^2 n/2$$

# Recurrences

## Substitution method

Another possible method is to make a **guess and then prove** the guess correctness using **induction**

- "Strong" vs "Weak" induction
  - ▶ With **weak induction** we assume it is valid for  $n$  and then we prove  $n + 1$
  - ▶ With **strong induction** we assume it is valid for all  $n_0 < n$  and we prove it for  $n$ .
- There are two "main" ways to use the substitution method:
  - ▶ We have an **exact guess**, with no "unknowns" (ex:  $3n^2 - n$ )
  - ▶ We only have **an idea of the class it belongs to** (ex:  $cn^2$ )
- How to prove that some  $f(n)$  is  $\Theta(g(n))$ ?
  - ▶ If we have an exact formula, just use it
  - ▶ Else, it may be "easier" to separately prove  $O$  and  $\Omega$ 
    - ★ Ex: to prove  $O$  we can show it is less than  $c.g(n)$
    - ★ Ex: to prove  $\Omega$  we can show it is more than  $c.g(n)$

# Recurrences

## Substitution method

”Prove that  $T(n) = T(n - 1) + n$  is  $\Theta(n^2)$ ”

Can we have an **exact guess**?

Let's assume  $T(1) = 1$

$$\begin{aligned}T(n) &= T(n - 1) + n \\&= T(n - 2) + (n - 1) + n \\&= T(n - 3) + (n - 2) + (n - 1) + n \\&= 1 + 2 + 3 + \dots + (n - 1) + n \\&= \frac{(n+1)n}{2} \text{ (An arithmetic progression)}\end{aligned}$$

# Recurrences

## Substitution method

**"Prove that  $T(n) = T(n-1) + n$  is  $\Theta(n^2)$ "**

Our (exact) guess is  $\frac{(n+1)n}{2}$

Now, let's try to prove by substituting.

Assuming it is true for  $n-1$ :

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= \frac{n(n-1)}{2} + n \\ &= \frac{n^2-n}{2} + n \\ &= \frac{n^2-n+2n}{2} \\ &= \frac{n^2+n}{2} \\ &= \frac{(n+1)n}{2} \quad \square \text{ (An we have proved our guess!)} \end{aligned}$$

# Recurrences

## Substitution method

**"Prove that  $T(n) = T(n/2) + 1$  is  $\Theta(\log_2 n)$ "**

And if we don't have an exact guess?

Let's try to prove that  $T(n) = \mathcal{O}(\log_2 n)$

We basically need to prove that  $T(n) \leq c \log_2 n$ , with  $n \geq n_0$ , for a correct choice of  $c$  and  $n_0$ .

Let's assume  $T(1) = 0$  and  $T(2) = 1$ . For these base cases:

- $T(1) \leq c \log_2 1$  for any  $c$ , because  $\log_2 1 = 0$
- $T(2) \leq c \log_2 2$  is true as long as  $c \geq 1$ .

Now, assuming it is true for all  $n' < n$ :

$$\begin{aligned} T(n) &\leq c \log_2(n/2) + 1 \\ &= c(\log_2 n - \log_2 2) + 1 \\ &= c \log_2 n - c + 1 \\ &\leq c \log_2 n, \text{ as long as } c \geq 1 \quad \square \text{ (We proved } T(n) = \mathcal{O}(\log_2 n)) \end{aligned}$$

# Recurrences

## Substitution method

"Prove that  $T(n) = T(n/2) + 1$  is  $\Theta(\log_2 n)$ "

Let's try to prove that  $T(n) = \Omega(\log_2 n)$

We basically need to prove that  $T(n) \geq c \log_2 n$ , with  $n \geq n_0$ , for a correct choice of  $c$  and  $n_0$ .

Let's assume  $T(1) = 0$  and  $T(2) = 1$ . For these base cases:

- $T(1) \geq c \log_2 1$  for any  $c$ , because  $\log_2 1 = 0$
- $T(2) \geq c \log_2 2$  is true as long as  $c \leq 1$ .

Now, assuming it is true for all  $n' < n$ :

$$\begin{aligned} T(n) &\geq c \log_2(n/2) + 1 \\ &= c(\log_2 n - \log_2 2) + 1 \\ &= c \log_2 n - c + 1 \\ &\geq c \log_2 n, \text{ as long as } c \leq 1 \quad \square \text{ (We proved } T(n) = \Omega(\log_2 n)) \end{aligned}$$

$T(n) = \mathcal{O}(\log_2 n)$  and  $T(n) = \Omega(\log_2 n) \rightarrow T(n) = \Theta(\log_2 n)$

# Solving Recurrences

## Substitution Method

If the guess is wrong, often we will gain clues for a better guess.

**Recurrence to solve:**  $T(n) = 4T(n/4) + n$

**Guess #1:**  $T(n) \leq cn$  (which would mean  $T(n) = \mathcal{O}(n)$ )

**Attempt to prove Guess #1:**

If  $T(1) = c$ , then the base case is true. For the rest of the induction, assuming it is true for  $n' < n$ , we can substitute using  $n' = n/4$ :

$$\begin{aligned} T(n) &\leq 4(cn/4) + n \\ &= cn + n \\ &= (c+1)n \end{aligned} \quad \text{but } (c+1)n \text{ is never } \leq cn \text{ for a positive } c$$

(the guess is wrong!)

We guess that we might need a higher function than simply  $\mathcal{O}(n)$

# Solving Recurrences

## Substitution Method

**Recurrence to solve:**  $T(n) = 4T(n/4) + n$

**Guess #2:**  $T(n) \leq n \log_4 n$

(I'm proving a more tight bound than simply  $cn \log_4 n$ )

**Attempt to prove Guess #2:**

If  $T(1) = 1$ , then the base case is true. For the rest of the induction, assuming it is true for  $n' < n$ , we can substitute using  $n' = n/4$ :

$$\begin{aligned} T(n) &\leq 4[(n/4) \log_4(n/4)] + n \\ &= n \log_4(n/4) + n \\ &= n \log_4(n) - n + n \\ &= n \log_4(n) \quad \square \text{ [correct guess! In fact, } T(n) = \Theta(n \log_4 n)] \end{aligned}$$



# Solving Recurrences

## Substitution Method - Subtleties

Sometimes you might correctly guess an asymptotic bound on the solution of a recurrence, but somehow the *math fails to work out in the induction*.

The problem frequently turns out to be that the **inductive assumption is not strong enough** to prove the detailed bound. If you **revise the guess by subtracting a lower-order term** when you hit such a snag, the math often goes through.

Let's observe an example of this:

**Recurrence to solve:**  $T(n) = 4T(n/2) + n$

As you will see later,  $T(n) = \Theta(n^2)$

Let's try to prove that directly.

# Solving Recurrences

## Substitution Method - Subtleties

**Recurrence to solve:**  $T(n) = 4T(n/2) + n$

**Guess #1:**  $T(n) \leq cn^2$

**Attempt to prove Guess #1:**

If  $T(1) = 1$ , then the base case is true as long as  $c \leq 1$ .

Now, assuming it is true for  $n' < n$

$$\begin{aligned} T(n) &\leq 4[c(n/2)^2] + n \\ &= cn^2 + n \quad \text{[which is not } \leq cn^2 \text{ for any positive } n\text{]} \end{aligned}$$

Although the bound is correct, the math does not work out...

We need a tighter bound to form a **stronger induction hypothesis**.

Let's **subtract a lower order-term** and try  $T(n) \leq c_1n^2 - c_2n$

# Solving Recurrences

## Substitution Method - Subtleties

**Recurrence to solve:**  $T(n) = 4T(n/2) + n$

**Guess #2:**  $T(n) \leq c_1 n^2 - c_2 n$

**Attempt to prove Guess #2:**

If  $T(1) = 1$ , then the base case is true as long as  $c_1 - c_2 \leq 1$

Now, assuming it is true for  $n' < n$

$$\begin{aligned} T(n) &\leq 4[c_1(n/2)^2 - c_2(n/2)] + n \\ &= c_1 n^2 - 2c_2 n + n \\ &= c_1 n^2 - c_2 n \quad \text{[correct guess!]} \end{aligned}$$

# Solving Recurrences

## Recursion Tree Method

Another method is to **draw a recursion tree** and analyse it, by summing all the work in the tree nodes.

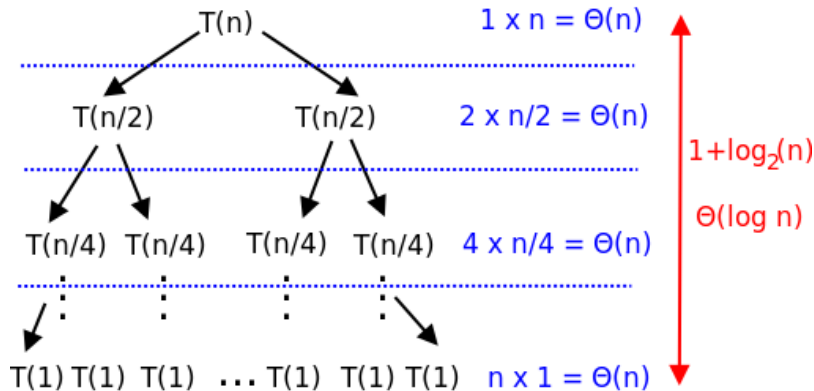
This method could be also used to get a good guess which we could then prove by induction.

Let us try it out with MergeSort:  $T(n) = 2(n/2) + n$

(for a cleaner explanation we will assume  $n = 2^k$ ,  
but the results holds for any  $n$ )

# Solving Recurrences

## Recursion Tree Method



Summing everything we get that **MergeSort** is  $\Theta(n \log_2 n)$

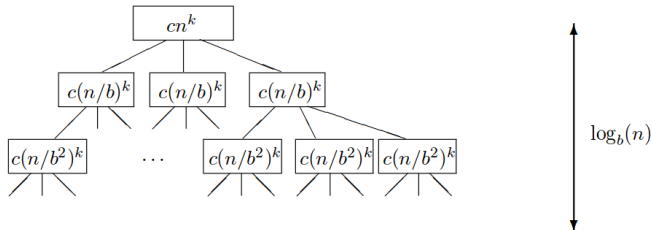
# Solving Recurrences

## Master Theorem

We can use the **master theorem** for recurrences of the following form:

$$T(n) = aT(n/b) + cn^k$$

This is well suited for divide and conquer recurrences and corresponds to an algorithm that divides the problem into **a** pieces of size **n/b** and takes **cn<sup>k</sup>** time for partitioning+combining.



In the mergesort case,  $a = 2$ ,  $b = 2$ ,  $k = 1$ .

## Ex. 3.7: Solve using recursion trees (assume $T(0)$ is a constant)

1.  $T(n) = k + T(n - 1)$  where  $k$  is a constant
2.  $T(n) = k + T(n/2)$  where  $k$  is a constant
3.  $T(n) = k + 2 * T(n/2)$  where  $k$  is a constant
4.  $T(n) = n + T(n - 1)$
5.  $T(n) = n + T(n/2)$
6.  $T(n) = n + 2 * T(n/2)$

**Ex. 3.8:** Write recurrences for `maxSumR` (wrt array accesses) and `Hanoi` (wrt `printf`)

**Ex. 3.9:** Draw a recurrence tree for `Hanoi` and use it to derive its asymptotic complexity

```
int maxSumR (int v[], int N) {
    int r=0, m1, m2, i;
    if (N>0) {
        m1 = m2 = v[0];
        for (i=1; i<N; i++) {
            m2 = m2+v[i];
            if (m2>m1) m1=m2;
        }
        m2 = maxSumR (v+1,N-1);
        if (m1>m2) r = m1; else r = m2;
    }
    return r; }
```

```
void Hanoi(int nDiscs, int l,
           int r, int m) {
    if (nDiscs > 0) {
        Hanoi(nDiscs-1, l, m, r);
        printf("move disk: %d-->%d\n", l, r);
        Hanoi(nDiscs-1, m, r, l);
    }
}
```



```
int heightBT(BTree t){  
    int r=0;  
    if (t!=NULL)  
        r = 1 + max (heightBT(t->left),  
                     heightBT(t->right));  
    return r;  
}
```

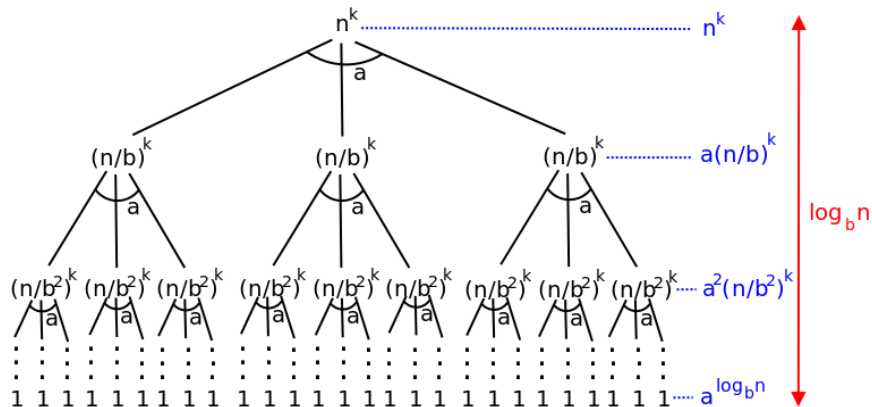
**Ex. 3.10:** Recall binary trees; this function calculates the maximum height of a binary tree. Identify the best and worst cases for this function, and describe a recurrence for each one.

slides by Pedro Ribeiro, slides 2  
pages 62-69

# Master Theorem

Intuition behind it

$aT(n/b) + n^k$  (I assume  $c = 1$  for a cleaner explanation)



# Master Theorem

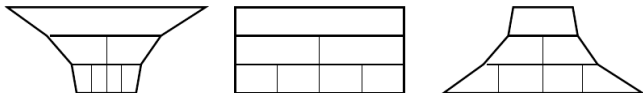
## Intuition behind it

- **Root (first level):**  $n^k$
- **Depth  $i$  (intermediate):**  $a^i (n/b^i)^k = a^i / b^{ik} n^k = (a/b^k)^i n^k$
- **Leafs (last level):**  $a^{\log_b n} = n^{\log_b a}$

So the weight of depth  $i$  is:  $(a/b^k)^i n^k$

- (1)  $a < b^k$  implies that  $a/b^k$  is lower than 1 (weight is shrinking)
- (2)  $a = b^k$  implies that  $a/b^k$  is equal to 1 (weight is constant)
- (3)  $a > b^k$  implies that  $a/b^k$  is higher than 1 (weight is growing)

- (1) The time is dominated by the **top level**
- (2) The time is (uniformly) **distributed** along the recursion tree
- (3) The time is dominated by the **last level**



# Master Theorem

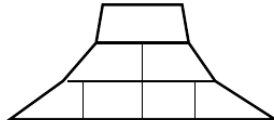
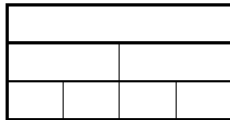
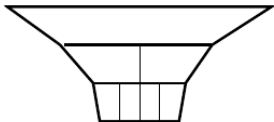
## Master Theorem - A practical version

A recurrence  $aT(n/b) + cn^k$  ( $a \geq 1, b > 1, c$  and  $k$  are constants) solves to:

- (1)  $T(n) = \Theta(n^k)$  if  $a < b^k$
- (2)  $T(n) = \Theta(n^k \log n)$  if  $a = b^k$
- (3)  $T(n) = \Theta(n^{\log_b a})$  if  $a > b^k$

If you think on the recursion tree, intuitively, these 3 cases correspond to:

- (1) The time is dominated by the **top level**
- (2) The time is (uniformly) **distributed** along the recursion tree
- (3) The time is dominated by the **last level**



# Master Theorem

## Master Theorem - A practical version

A recurrence  $aT(n/b) + cn^k$  ( $a \geq 1, b > 1, c$  and  $k$  are constants) solves to:

- (1)  $T(n) = \Theta(n^k)$  if  $a < b^k$
- (2)  $T(n) = \Theta(n^k \log n)$  if  $a = b^k$
- (3)  $T(n) = \Theta(n^{\log_b a})$  if  $a > b^k$

### Example of Case (1):

$$T(n) = 2T(n/2) + n^2$$

$a = 2, b = 2, k = 2, a < b^k$  since  $2 < 4$ .

The recurrence solves to  $\Theta(n^2)$

# Master Theorem

## Master Theorem - A practical version

A recurrence  $aT(n/b) + cn^k$  ( $a \geq 1, b > 1, c$  and  $k$  are constants) solves to:

- (1)  $T(n) = \Theta(n^k)$  if  $a < b^k$
- (2)  $T(n) = \Theta(n^k \log n)$  if  $a = b^k$
- (3)  $T(n) = \Theta(n^{\log_b a})$  if  $a > b^k$

### Example of Case (2):

$$T(n) = 2T(n/2) + n \text{ (ex: mergesort)}$$

$$a = 2, b = 2, k = 1, a = b^k \text{ since } 2 = 2.$$

The recurrence solves to  $\Theta(n \log n)$  (as we already knew).

# Master Theorem

## Master Theorem - A practical version

A recurrence  $aT(n/b) + cn^k$  ( $a \geq 1, b > 1, c$  and  $k$  are constants) solves to:

- (1)  $T(n) = \Theta(n^k)$  if  $a < b^k$
- (2)  $T(n) = \Theta(n^k \log n)$  if  $a = b^k$
- (3)  $T(n) = \Theta(n^{\log_b a})$  if  $a > b^k$

### Example of Case (3):

$$T(n) = 2T(n/2) + 1$$

$a = 2, b = 2, k = 0, a > b^k$  since  $2 > 1$ .

The recurrence solves to  $\Theta(n)$



# Master Theorem

## Revisiting the examples

Examples:

$$(1) \quad T(n) = 2T(n/2) + n^2 = \Theta(n^2)$$

$n^2 + n^2/2 + n^2/4 + \dots + n \leftarrow (n^2 \text{ dominates, i.e., the root})$

$$(2) \quad T(n) = 2T(n/2) + n = \Theta(n \log n)$$

$n + n + \dots + n \leftarrow (\text{distributed among all levels})$

$$(3) \quad T(n) = 2T(n/2) + 1 = \Theta(n)$$

$1 + 2 + 4 + \dots + n \leftarrow (n \text{ dominates, i.e., the leaf})$

# Master Theorem

For the sake of completeness, here is the master theorem version presented in the book "**Introduction to Algorithms**".

## Master Theorem

A more general version A recurrence  $aT(n/b) + f(n)$  ( $a \geq 1, b > 1$  are constants) solves to:

- (1) If  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$  for some constant  $\epsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$
- (2) If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$
- (3) If  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for some constant  $\epsilon > 0$ , and if  $af(n/b) \leq cf(n)$  for some constant  $c < 1$  and all sufficiently large  $n$ , then  $T(n) = \Theta(f(n))$

(cases 1 and 3 are inverted in relation to the practical version I've shown)

slides by Pedro Ribeiro, exercises 3  
pages 1-2

## Exercises #3

### Solving Recurrences

---

#### Theoretical Background

4 methods for solving recurrences:

- **Unrolling:** unroll the recurrence to obtain an expression (ex: summation) you can work with
  - **Substitution:** guess the answer and prove by induction
  - **Recursion Tree:** draw a tree representing the recursion and sum all the work done in the nodes
  - **Master Theorem:** If the recurrence is of the form  $aT(n/b) + cn^k$  (*this is one version of the theorem*):
    - (1)  $T(n) = \Theta(n^k)$  if  $a < b^k$
    - (2)  $T(n) = \Theta(n^k \log n)$  if  $a = b^k$
    - (3)  $T(n) = \Theta(n^{\log_b a})$  if  $a > b^k$
- 

For the following exercises, assume that  $T(n)$  takes constant time for sufficiently small  $n$ .

1. Solve the following recurrences by unrolling. State the answer using  $\Theta$  notation.
  - (a)  $T(n) = T(n-2) + 1$
  - (b)  $T(n) = T(n-1) + n^2$
2. Show that the following conjectures are true by using the substitution method.
  - (a)  $T(n) = T(n-1) + 2$  is  $\Theta(n)$
  - (b)  $T(n) = 2T(n/2) + n$  is  $\Theta(n \log n)$
3. Draw a recursion tree for the following recurrences and use it to obtain asymptotic bounds as tight as possible.
  - (a)  $T(n) = 3T(n/2) + n$
  - (b)  $T(n) = T(n/2) + n^2$
4. Solve the following recurrences using the master method:
  - (a)  $T(n) = 2T(n/4) + 1$
  - (b)  $T(n) = 2T(n/4) + \sqrt{n}$
  - (c)  $T(n) = 2T(n/4) + n$
  - (d)  $T(n) = 2T(n/4) + n^2$

5. Consider the recurrence  $T(n) = 8T(n/2) + n^2$
- (a) Use the substitution method to try to prove that  $T(n) = O(n^3)$ . The proof should fail. Can you understand why?
  - (b) Use the master method to find the a tight asymptotic bound. Try to prove that bound directly. Does the math work?
  - (c) Use a stronger induction hypothesis (by subtracting a lower order term) and make a correct proof of that tighter bound.
6. Give asymptotic upper and lower bounds (as tight as possible) for the following recurrences. You can use any method you want.
- (a)  $T(n) = 7T(n/3) + n^2$
  - (b)  $T(n) = 7T(n/2) + n^2$
  - (c)  $T(n) = 2T(n/4) + n^2$
  - (d)  $T(n) = T(n-2) + n^3$
  - (e)  $T(n) = T(n/2) + T(n/4) + T(n/8) + n$
  - (f)  $T(n) = T(n-1) + \frac{1}{n}$
  - (g)  $T(n) = 4T(n/3) + n \log_2 n$