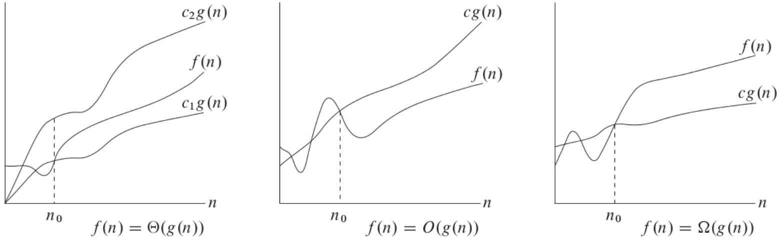
# Asymptotic Analysis

Pedro Ribeiro

DCC/FCUP

2018/2019

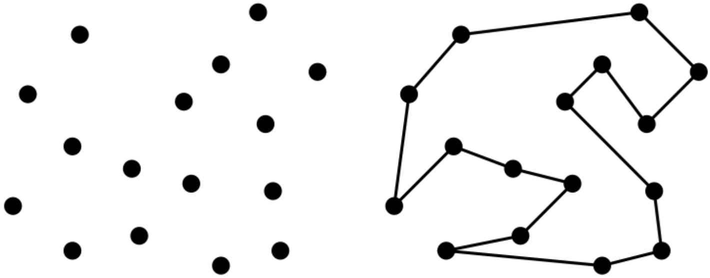


# Motivational Example - TSP

**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:



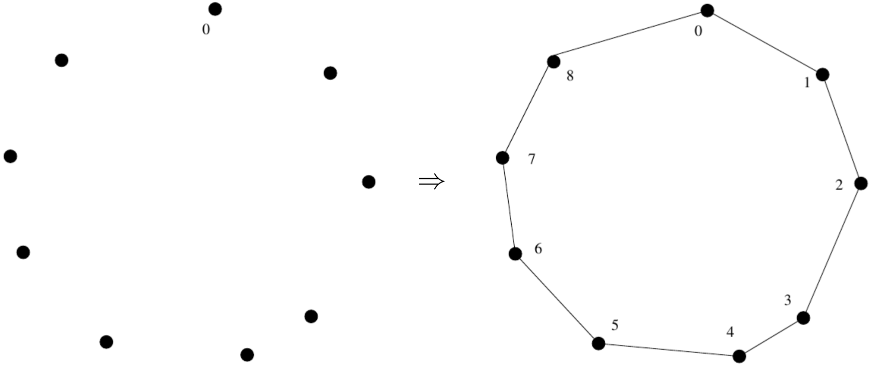
# Motivational Example - TSP

**A possible (greedy) algorithm - nearest neighbour**

$p_1 \leftarrow$ random point
$i \leftarrow 1$
**While** (there are still points to visit) **do**
    $i \leftarrow i + 1$
    $p_i \leftarrow$ nearest non visited neighbour of point $p_{i-1}$
**return** path $p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n \rightarrow p_1$

# Motivational Example - TSP

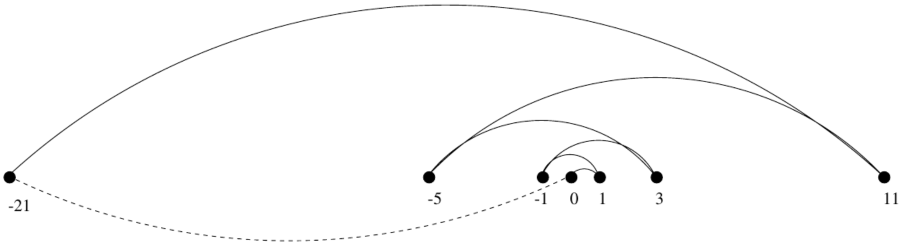
Seems to work...

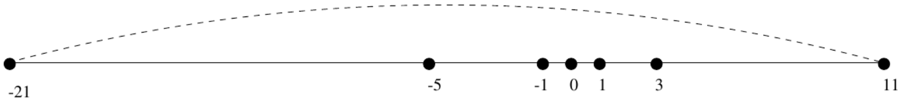


# Motivational Example - TSP

But it is does not work for all instances!

(Note: starting with the leftmost point would not solve the problem)



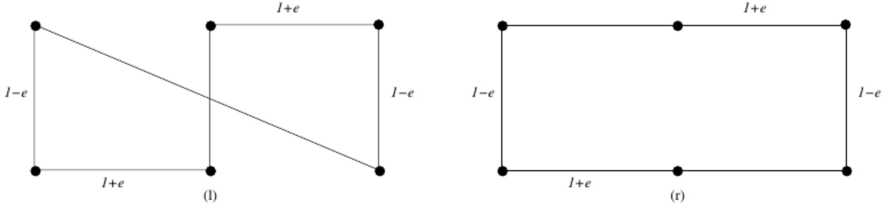
# Motivacional Example - TSP

**Another possible (greedy) algorithm**

**For** $i \leftarrow 1$ **to** $(n-1)$ **do**
    Add connection between closest pair of points such that
    they are in different connected components
Add connection between the two "extremes" of the created path
**return** the cyclic path created

# Motivational Example - TSP

It does not work for all cases!



# 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**
        $P_{min} \leftarrow P_i$
**retorn** Path formed by $P_{min}$

A correct algorithm, but extremely slow!

- $P(n) = n! = n \times (n-1) \times \ldots \times 1$
- 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 min |
| $N^4$ | $0.1s$ | 16 min | 115 days |
| $2^N$ | $10^{13}$ years | $10^{284}$ years | $10^{2993}$ years |
| $n!$ | $10^{141}$ years | $10^{2551}$ years | $10^{35642}$ 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:
  $P(n) = n! = n \times (n-1) \times \ldots \times 1$
  (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**)

$\mathbf{n} \leq \mathbf{7}$: $< 0.001s$
$\mathbf{n} = \mathbf{8}$: $0.001s$
$\mathbf{n} = \mathbf{9}$: $0.016s$
$\mathbf{n} = \mathbf{10}$: $0.185s$
$\mathbf{n} = \mathbf{11}$: $2.204s$
$\mathbf{n} = \mathbf{12}$: $28.460s$
...
$\mathbf{n} = \mathbf{20}$: 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

**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 $2n$ |

### 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 $n$

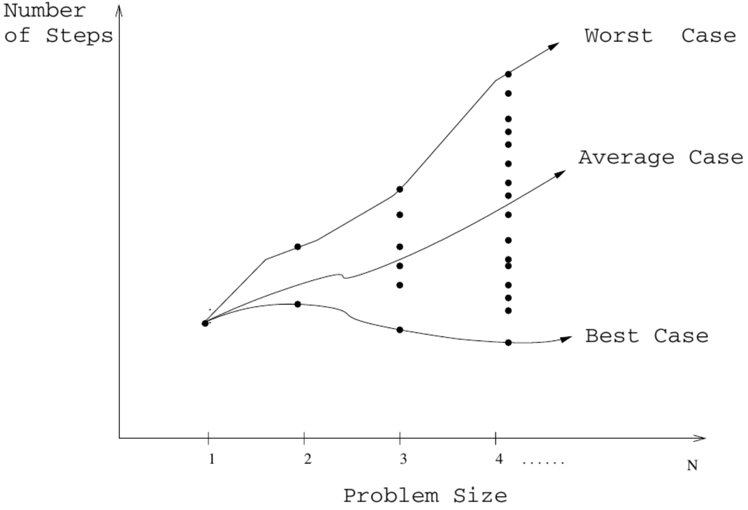
**Average Case** analysis: **(sometimes)**

- $T(n) =$ 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

# 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 $n \to \infty$)
- 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

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

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

### $\mathbf{f(n) = \Omega(g(n))}$

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

### $\mathbf{f(n) = \Theta(g(n))}$

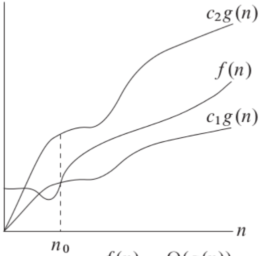
It means that $c_1 \times g(n)$ is a **lower bound** of $f(n)$ and $c_2 \times g(n)$ is an **upper bound** of $f(n)$

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

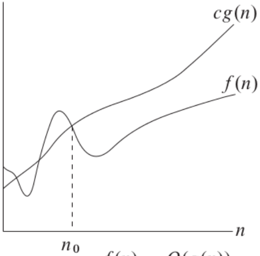
# Asymptotic Notation

## A graphical depiction

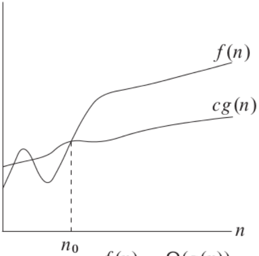
| $\Theta$ | $\mathcal{O}$ | $\Omega$ |
| --- | --- | --- |




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




$f(n) = O(g(n))$




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

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

- $\mathbf{f(n)} = \mathcal{O}(\mathbf{g(n)})$ if there exist positive constants $n_0$ and $c$ such that $f(n) \leq c \times g(n)$ for all $n \geq n_0$
- $\mathbf{f(n)} = \mathbf{\Omega}(\mathbf{g(n)})$ if there exist positive constants $n_0$ and $c$ such that $f(n) \geq c \times g(n)$ for all $n \geq n_0$
- $\mathbf{f(n)} = \mathbf{\Theta}(\mathbf{g(n)})$ if there exist positive constants $n_0$, $c_1$ and $c2$ such that $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ for all $n \geq n_0$

- $\mathbf{f(n)} = \mathbf{o}(\mathbf{g(n)})$ if for any positive constant $c$ there exists $n_0$ such that $f(n) < c \times g(n)$ for all $n \geq n_0$
- $\mathbf{f(n)} = \omega(\mathbf{g(n)})$ if for any positive constant $c$ there exists $n_0$ such that $f(n) > c \times g(n)$ for all $n \geq n_0$

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 |
| $f(n) = \mathbf{\Omega}(g(n))$ | is like | $a \geq b$ | lower bound | at least as bad as |
| $f(n) = \mathbf{\Theta}(g(n))$ | 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 $f(n) = \Omega(g(n))$
- $f(n) = \mathcal{O}(g(n)) \rightarrow f(n) \neq \omega(g(n))$
- $f(n) = \Omega(g(n)) \rightarrow f(n) \neq \mathbf{o}(g(n))$
- $f(n) = \mathbf{o}(g(n)) \rightarrow f(n) \neq \Omega(g(n))$
- $f(n) = \omega(g(n)) \rightarrow f(n) \neq \mathcal{O}(g(n))$

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

# 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 $a_x n^x + a_{x-1} n^{x-1} + \ldots + a_2 n^2 + a_1 n + a_0$ we can focus on the term with the **largest exponent**:
  $3\mathbf{n^3} - 5n^2 + 100 = \Theta(n^3)$
  $6\mathbf{n^4} - 20^2 = \Theta(n^4)$
  $0.8\mathbf{n} + 224 = \Theta(n)$

- More than that, on a sum we can focus on the **dominant** term:
  $\mathbf{2^n} + 6n^3 = \Theta(2^n)$
  $\mathbf{n}! - 3n^2 = \Theta(n!)$
  $n \log n + 3\mathbf{n^2} = \Theta(n^2)$

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 $\lim_{n\to\infty} g(n)/f(n) = 0$

**Dominance Relations**

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

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

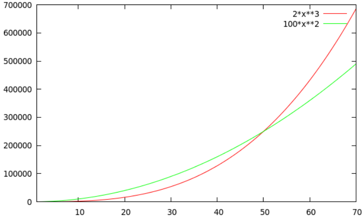
## Common Functions

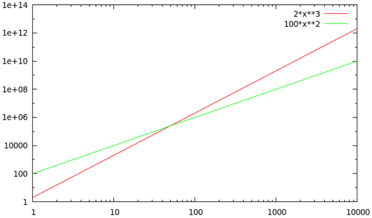
| Function | Name | Examples |
|---|---|---|
| $1$ | constant | summing two numbers |
| $\log n$ | logarithmic | binary search, inserting in a heap |
| $n$ | linear | 1 loop to find maximum value |
| $n \log n$ | linearithmic | sorting (ex: mergesort, heapsort) |
| $n^2$ | quadratic | 2 loops (ex: verifying,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

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 [1:1000000] sqrt(x), log(x)/log(2)
```

- 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 $n$.

# Analyzing the complexity of programs

Let's see more concrete examples:

- **Case 1 Loops** (and summations)
- **Case 2 Recursive Functions** (and recurrences)

# Loops and Summations

**A typical loop**

$count \leftarrow 0$
**For** $i \leftarrow 1$ **to** $1000$
  **For** $j \leftarrow i$ **to** $1000$
    $count \leftarrow count + 1$
**write**($count$)

What does this program write?

$1000 + 999 + 998 + 997 + \ldots.. + 2 + 1$

# Loops and Summations

**Arithmetic progression**: a sequence of numbers such that the difference $d$ between the consecutive terms is constant. We will call $a_1$ to the first term.

- $1, 2, 3, 4, 5, \ldots \ldots$ $(d = 1, a_1 = 1)$
- $3, 5, 7, 9, 11, \ldots \ldots$ $(d = 2, a_1 = 3)$

How to calculate the summation of an arithmetic progression?

$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = (1 + 8) + (2 + 7) + (3 + 6) + (4 + 5) = 4 \times 9$

**Summation from $a_p$ to $a_q$**

$$S(p, q) = \sum_{i=p}^{q} a_i = \frac{(q-p+1) \times (a_p + a_q)}{2}$$

**Summation of the first $n$ terms**

$$S_n = \sum_{i=1}^{n} a_i = \frac{n \times (a_1 + a_n)}{2}$$

# Loops and Summations

## A typical loop

$count \leftarrow 0$
**For** $i \leftarrow 1$ **to** $1000$
  **For** $j \leftarrow i$ **to** $1000$
    $count \leftarrow count + 1$
**write**($count$)

What does this program write?

$1000 + 999 + 998 + 997 + \ldots.. + 2 + 1$

It writes $S_{1000} = \frac{1000 \times (1000+1)}{2} = 500500$

# Loops and Summations

## A typical loop

$count \leftarrow 0$
**For** $i \leftarrow 1$ **to** $n$
    **For** $j \leftarrow i$ **to** $n$
        $count \leftarrow count + 1$
**write**($count$)

What is the execution time?

It is going to execute $S_n$ increments:

$$S_n = \sum_{i=1}^{n} a_i = \frac{n \times (1+n)}{2} = \frac{n+n^2}{2} = \frac{1}{2}n^2 + \frac{1}{2}n.$$

It executes $\Theta(n^2)$ steps

# Loops and Summations

If you want to know more about interesting summations on the context of CS, take a look at *Appendix A* of the *Introduction to Algorithms* book.

Note that $c$ loops do not imply $\Theta(n^c)$!

**A loop**

```
For i ← 1 to n
    For j ← 1 to 5
```

$\Theta(n)$

**Another loop**

```
For i ← 1 to n
    For j ← 1 to i × i
```

$\Theta(n^3)$ $(1^2 + 2^2 + 3^2 + \ldots + n^2 = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

# 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

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

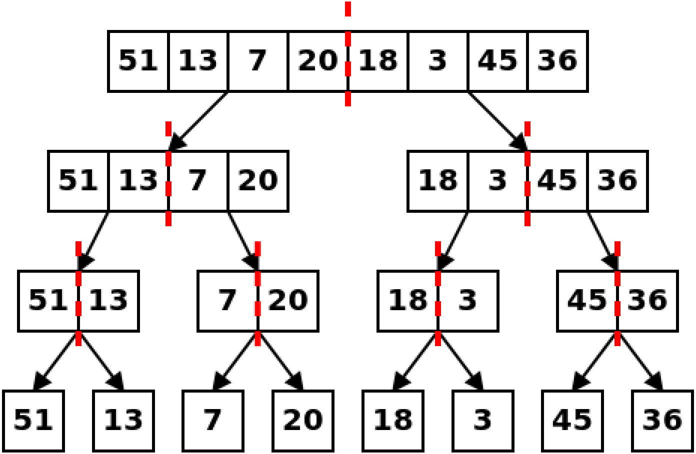
### MergeSort

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

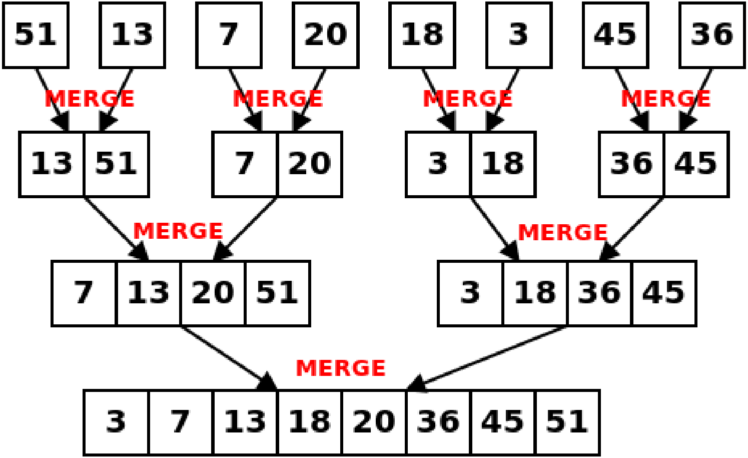
**Conquer:** recursively sort each half. If we only have one number, it is sorted.

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

## Divide:


51 13 7 20 18 3 45 36
51 13 7 20
18 3 45 36
51 13
7 20
18 3
45 36
51
13
7
20
18
3
45
36

**Conquer:**


51
13
7
20
18
3
45
36
MERGE
MERGE
MERGE
MERGE
13 51
7 20
3 18
36 45
MERGE
MERGE
7 13 20 51
3 18 36 45
MERGE
3 7 13 18 20 36 45 51

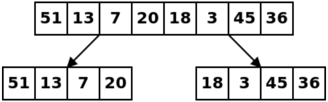
What is the **execution time** of this algorithm?

- **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 $n/2$)

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



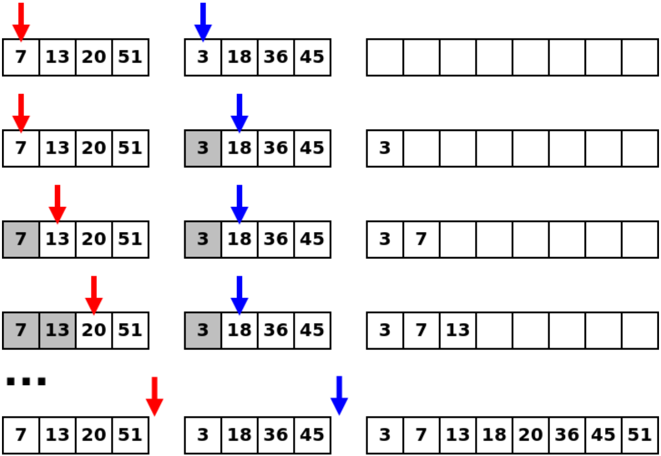

We can do it in constant time! $\mathbf{\Theta(1)}$

`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)`

$\mathbf{M(n)}$ - Time to merge two sorted arrays of size $n$




We can do it in linear time! $\color{red}{\mathbf{\Theta(n)}}$ ($2n$ comparisons)

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 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?

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 $\mathbf{aT(n/b) + cn^k}$, the answer follows a certain pattern

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

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

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

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

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 $c.g(n)$

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

Can we have an **exact guess**?

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

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

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

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

Now, let's try to prove by substituting.

Assuming it is true for $n-1$:

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

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

And if we don't have an exact guess?

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

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 $n' < n$:

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

# Recurrences

## Substitution method

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

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

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 $n' < n$:

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

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

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 $n' = n/4$:

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

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

**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 $n' = n/4$:

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

# 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.

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

**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 $n' < n$

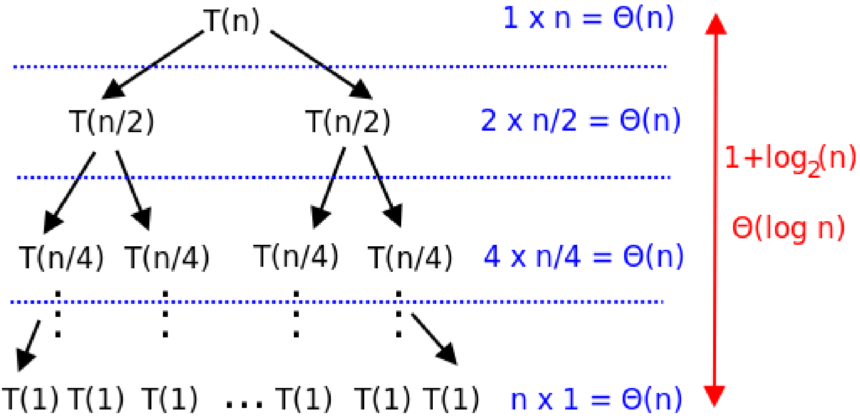
$$\begin{aligned} T(n) &\leq 4[c_1(n/2)^2 - c_2(n/2)] + n \\ &= c_1 n^2 - 2c_2 n + n \\ &= c_1 n^2 - c_2 n \quad \text{[correct guess!]} \end{aligned}$$

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 $n$)

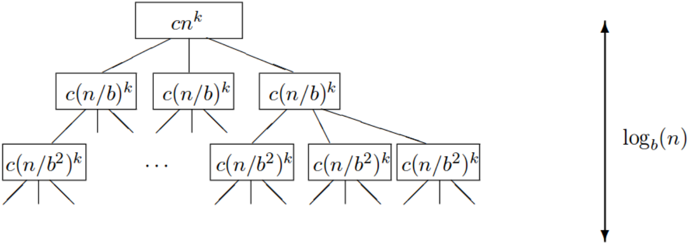



Summing everything we get that **MergeSort** is $\mathbf{\Theta(n \log_2 n)}$

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

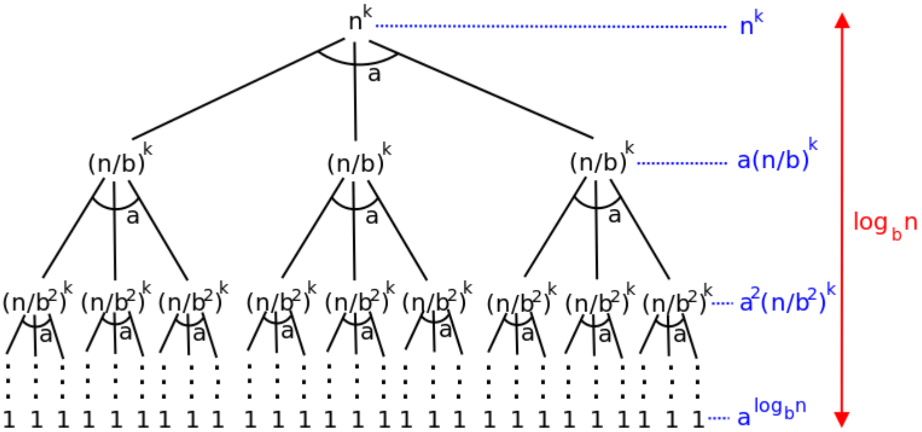
$$\mathbf{T(n) = aT(n/b) + cn^k}$$

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 $\mathbf{cn^k}$ time for partitioning+combining.




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

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



# Master Theorem

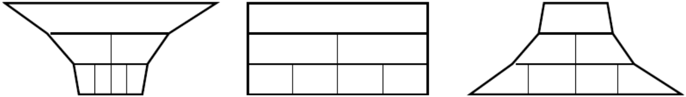
## Intuition behind it

- **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: $\mathbf{(a/b^k)^i n^k}$

| | | | |
|---|---|---|---|
| (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

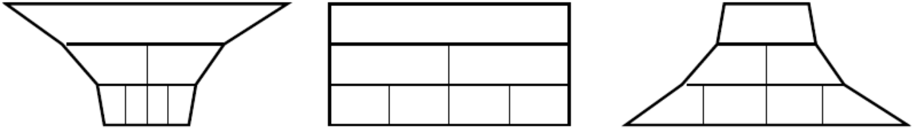
**Master Theorem - A practical version**

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

$$
\begin{array}{lll}
(1) & T(n) = \Theta(n^k) & \text{if } a < b^k \\
(2) & T(n) = \Theta(n^k \log n) & \text{if } a = b^k \\
(3) & T(n) = \Theta(n^{\log_b a}) & \text{if } a > b^k
\end{array}
$$

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

$$\begin{array}{lll} (1) & T(n) = \Theta(n^k) & \text{if } a < b^k \\ (2) & T(n) = \Theta(n^k \log n) & \text{if } a = b^k \\ (3) & T(n) = \Theta(n^{\log_b a}) & \text{if } a > b^k \end{array}$$

**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 $\mathbf{\Theta(n^2)}$

# Master Theorem

**Master Theorem - A practical version**

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

$$\begin{array}{lll} (1) & T(n) = \Theta(n^k) & \text{if } a < b^k \\ (2) & T(n) = \Theta(n^k \log n) & \text{if } a = b^k \\ (3) & T(n) = \Theta(n^{\log_b a}) & \text{if } a > b^k \end{array}$$

**Example of Case (2)**:

$T(n) = 2T(n/2) + n$ *(ex: mergesort)*

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

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

# Master Theorem

**Master Theorem - A practical version**

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

$$
\begin{array}{lll}
(1) & T(n) = \Theta(n^k) & \text{if } a < b^k \\
(2) & T(n) = \Theta(n^k \log n) & \text{if } a = b^k \\
(3) & T(n) = \Theta(n^{\log_b a}) & \text{if } a > b^k
\end{array}
$$

**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 $\mathbf{\Theta(n)}$

Examples:

(1) $T(n) = 2T(n/2) + n^2 = \Theta(n^2)$
$n^2 + n^2/2 + n^2/4 + \ldots + n \leftarrow$ ($n^2$ dominates, i.e., the root)

(2) $T(n) = 2T(n/2) + n = \Theta(n \log n)$
$n + n + \ldots + n \leftarrow$ (distributed among all levels)

(3) $T(n) = 2T(n/2) + 1 = \Theta(n)$
$1 + 2 + 4 + \ldots + n \leftarrow$ ($n$ 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 $\mathbf{aT(n/b) + f(n)}$ ($a \geq 1, b > 1$ 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 $T(n) = \Theta(f(n))$

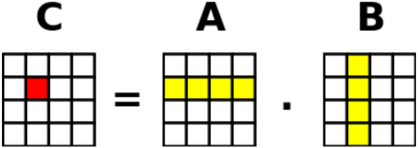
(cases 1 and 3 are inverted in relation to the practical version I've shown)

### Matrix Multiplication Problem

**Input:** Two $n \times n$ square matrices $A = (a_{ij})$ and $B = (b_{ij})$
**Output:** Compute $C = A \cdot B$

Remember matrix multiplication?

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$



### Naive Algorithm

**For** $i = 1$ **to** $n$
    **For** $j = 1$ **to** $n$
        $c_{ij} = 0$
        **For** $k = 1$ **to** $n$
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$

The complexity is $\mathbf{\Theta(n^3)}$

Suppose we partition each matrix in four:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then:

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

## Simple D&C Algorithm

**multiply**($A$,$B$)

$n = A.\text{number\_rows}$

$C = \text{new } x \times n \text{ matrix}$

**If** $n == 1$

$C_{11} = A_{11} \cdot B_{11}$

**Else**

Partition $A$, $B$ and $C$ as shown on the previous slide

$C_{11} = \textbf{multiply}(A_{11}, B_{11}) + \textbf{multiply}(A_{12}, B_{21})$

$C_{12} = \textbf{multiply}(A_{11}, B_{12}) + \textbf{multiply}(A_{12}, B_{22})$

$C_{21} = \textbf{multiply}(A_{21}, B_{11}) + \textbf{multiply}(A_{22}, B_{21})$

$C_{22} = \textbf{multiply}(A_{21}, B_{12}) + \textbf{multiply}(A_{22}, B_{22})$

Time to sum 4 squares of size $n/2$: $\Theta(n^2)$ Recurrence:
$\mathbf{T(n) = 8T(n/2) + cn^2}$ (by master theorem this is $\mathbf{\Theta(n^3)}$)

**Strassen's algorithm** (key idea: less recursive calls)

$S_1 = B_{12} - B_{22}$, $S_2 = A_{11} + A_{12}$, $S_3 = A_{21} + A_{22}$, $S_4 = B_{21} - B_{11}$, $S_5 = A_{11} + A_{22}$,
$S_6 = B_{11} + B_{22}$, $S_7 = A_{12} - A_{22}$, $S_8 = B_{21} + B_{22}$, $S_9 = A_{11} - A_{21}$, $S_10 = B_{11} + B_{12}$
10x add/subtract matrices of size $n/2$: $\mathbf{\Theta(n^2)}$

$P1 = A_{11} \cdot S_1$, $P2 = S_2 \cdot B_{22}$, $P3 = S_3 \cdot B_{11}$
$P4 = A_{22} \cdot S_4$, $P5 = S_5 \cdot S_6$, $P6 = S_7 \cdot S_8$
$P7 = S_9 \cdot S_{10}$
7 multiplications of matrices of size $n/2$: $\mathbf{7T(n/2)}$

$C_{11} = P_5 + P_4 - P_2 + P_6$
$C_{12} = P_1 + P_2$
$C_{21} = P_3 + P_4$
$C_{22} = P_5 + P_1 - P_3 - P_7$

8x add/subtract matrices of size $n/2$: $\mathbf{\Theta(n^2)}$

Recurrence: $\mathbf{T(n) = 7T(n/2) + cn^2}$
(by master theorem this is $\mathbf{\Theta(n^{\log_2 7})} \sim \mathbf{\Theta(n^{2.81})}$)

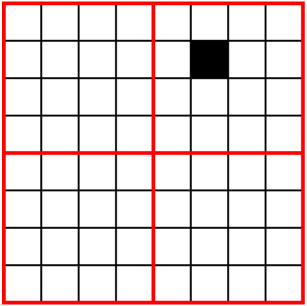
**Strassen's algorithm:**

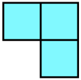
- Trade one recursion per constant additions/subtractions
- The "hidden " constant factor is larger than for the naive $\Theta(n^3)$
  For small inputs, the naive may be better.
- For sparse matrices, we could do other kind of optimizations
- The algorithm may have problem in numerical stability: the limited precision of floating point numbers in computers may accumulate errors.
- The intermediate submatrices consume memory space

- V Strassen, **Gaussian elimination is not optimal**, Numerische Mathematik 13 (1969). $\rightarrow \Theta(n^{2.81})$
- D Coppersmith, S. Winograd, **Matrix multiplication via arithmetic progressions**, Journal of Symbolic Computation 9 (3): 251 (1990). $\rightarrow \Theta(n^{2.375477})$
- AJ Stothers, **On the complexity of matrix multiplication**, PhD thesis, University of Edinburgh (2010). $\rightarrow \Theta(n^{2.373})$
- VV Williams. **Breaking the Coppersmith-Winograd barrier**, STOC'2012: Proceedings of the 44th annual ACM symposium on Theory of Computing, New York, USA, ACM Press (2012). $\rightarrow \Theta(n^{2.372873})$
- F Le Gall, **"Powers of tensors and fast matrix multiplication"**, ISSAC'2014, Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (2014). $\rightarrow \Theta(n^{2.3728639})$

The D&C can even be used... "manually" :)

Imagine a grid of size $2^n \times 2^n$. You want to **fill all cells with trominoes** (l-shaped pieces).
Pieces can be rotated and the initial grid has one cell which is "forbidden".

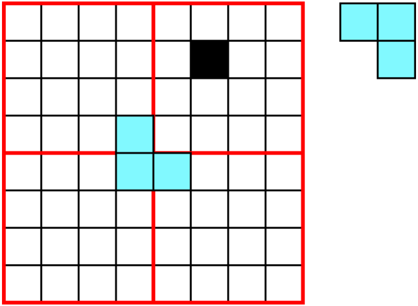




One idea is to **divide in 4 smaller squares**... and use one piece!

The D&C can even be used... "manually" :)

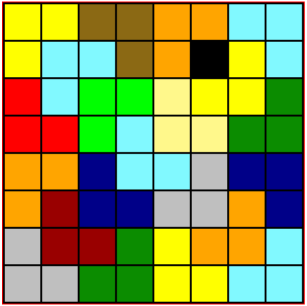
Imagine a grid of size $2^n \times 2^n$. You want to **fill all cells with trominoes** (L-shaped pieces).
Pieces can be rotated and the initial grid has one cell which is "forbidden".

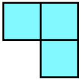


One idea is to **divide in 4 smaller squares**... and use one piece!

The D&C can even be used... "manually" :)

Imagine a grid of size $2^n \times 2^n$. You want to **fill all cells with trominoes** (l-shaped pieces).
Pieces can be rotated and the initial grid has one cell which is "forbidden".





One idea is to **divide in 4 smaller squares**... and use one piece!