

3. Counting steps (Asymptotic analysis) [WiP]

José Proença

Algorithms (CC4010) 2023/2024

CISTER – U.Porto, Porto, Portugal

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



- Measuring **precisely** performance of algorithms
- Measuring **asymptotically** performance of algorithms
- Measuring precisely the **average time** of algorithms
- Next: beyond worst-/best-case scenarios
 - Analysing **recursive** functions
 - 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”

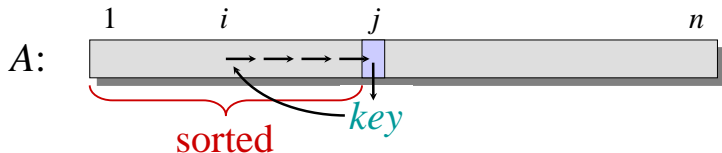
```
INSERTION-SORT ( $A, n$ )    ▷  $A[1 \dots n]$ 
  for  $j \leftarrow 2$  to  $n$ 
    do  $key \leftarrow A[j]$ 
       $i \leftarrow j - 1$ 
      while  $i > 0$  and  $A[i] > key$ 
        do  $A[i+1] \leftarrow A[i]$ 
           $i \leftarrow i - 1$ 
       $A[i+1] = key$ 
```




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



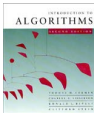
Example of insertion sort





Example of insertion sort





Example of insertion sort



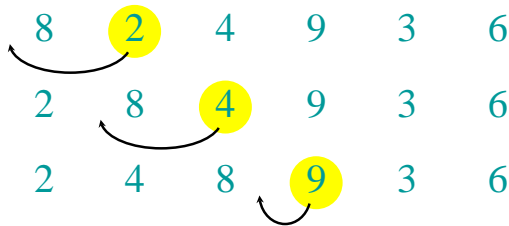


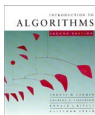
Example of insertion sort





Example of insertion sort





Example of insertion sort



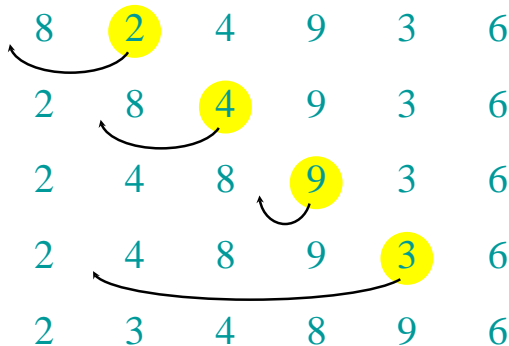


Example of insertion sort



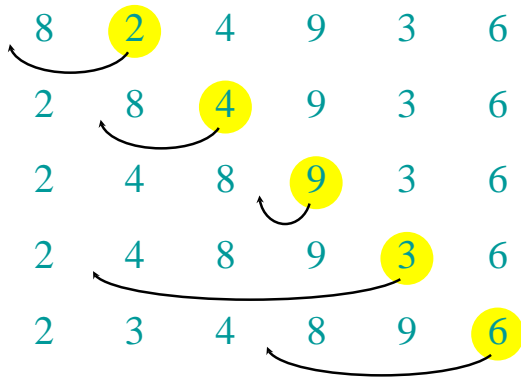


Example of insertion sort





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.

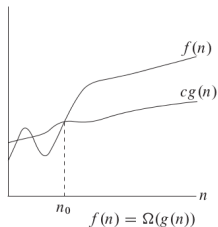
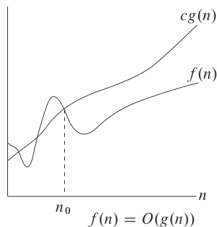
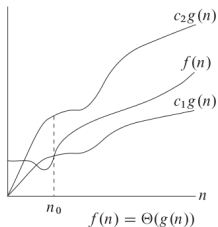
slides by Pedro Ribeiro, slides 2
pages 1-2

Asymptotic Analysis

Pedro Ribeiro

DCC/FCUP

2018/2019



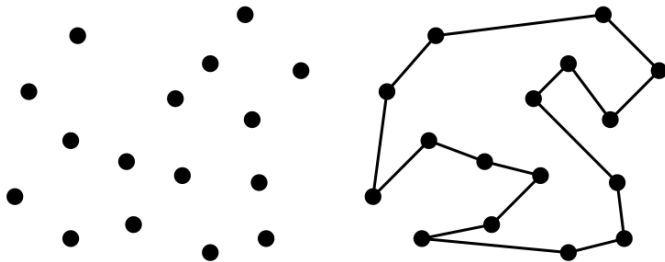
Motivational Example - TSP

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 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:

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

(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;  
}
```

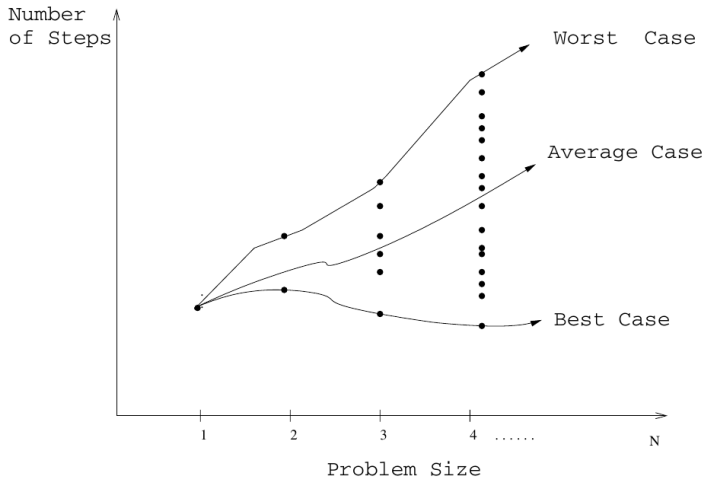
Ex. 3.5: What is the best and worst case for maxgrow wrt comparisons of array elements?

Ex. 3.6: How many comparisons 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 $f(n)$

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

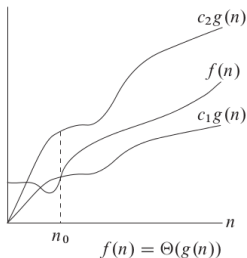
It means that $c_1 \times g(n)$ is a **lower bound** of $f(n)$ and $c_2 \times g(n)$ is an **upper bound** of $f(n)$

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

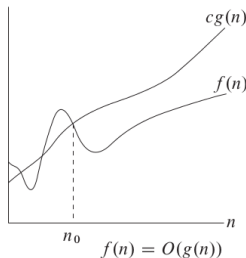
Asymptotic Notation

A graphical depiction

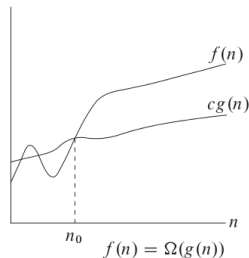
Θ



O



Ω



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

Asymptotic Notation

Formalization

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

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

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

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

Big Omega (Ω)

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

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

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

Big Theta (Θ)

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

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

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

Examples

Big Oh (O)

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

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

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

Big Omega (Ω)

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

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

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

Big Theta (Θ)

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

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

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

Examples

Big Oh (O)

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

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

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

Big Omega (Ω)

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

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

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

Big Theta (Θ)

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

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

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

Examples

Big Oh (O)

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

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

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

Big Omega (Ω)

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

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

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

Big Theta (Θ)

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

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

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

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

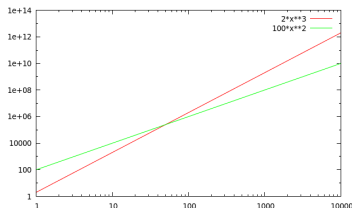
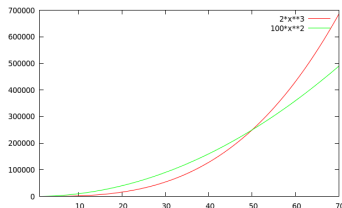
An useful program for drawing functions is **gnuplot**.

(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:

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

Asymptotic Notation

- Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$. Justify your answer with brief proofs.
- For each pair of functions $f(n)$ and $g(n)$, indicate whether $f(n)$ is O , o , Ω , ω , or Θ 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	Ω	ω	Θ
(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?

Average case

...

Recall arithmetic series

...

Recall geometric series

...

Recursive functions

[example with merge sort; avoid substitution method (or glimpse for a given guess);
focus on recurrent trees]

[exercises both from Ribeiro and JBB]

[Later: simplified version of master theorem and short exercises]