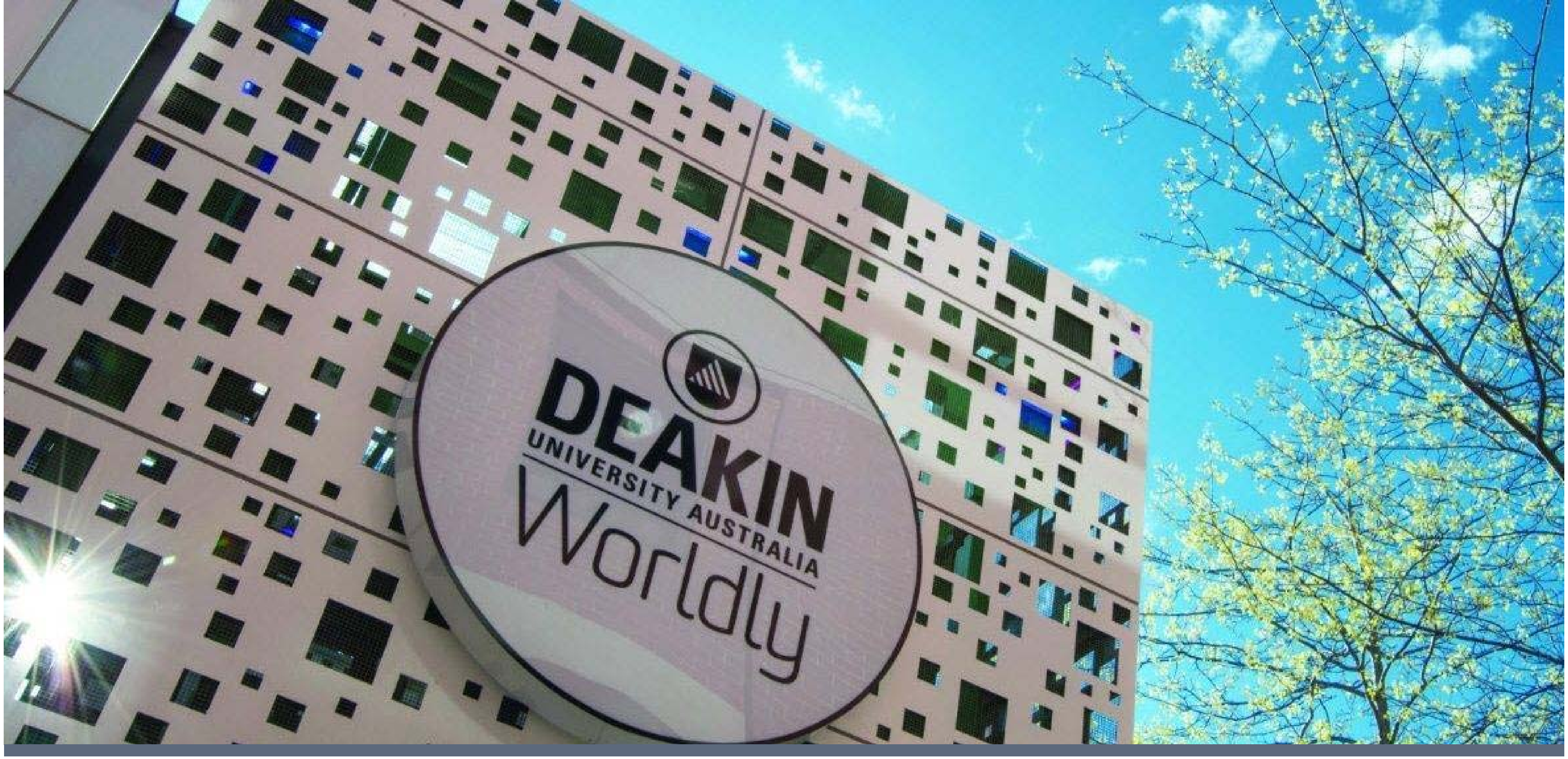


# Lecture 1. Introduction to Complexity of Algorithms. Asymptotic Notations

SIT221 Data Structures and Algorithms

## Our goals

**We want to have:**

- Data structures that allow us to carry out operations as efficiently as possible.
- Algorithms that solve problems as efficiently as possible.

## What is an Algorithm?

Many definitions! Here are a few:

- A self-contained step-by-step set of operations to be performed.
- A set of rules that **precisely** defines a sequence of operations.
- A set of steps to be followed in order to **solve** a problem.
- An algorithm is a **well-defined** sequence of steps used to **solve** a **well-defined** problem in **finite time**.

## What is an Algorithm?

- An **algorithm** is a **step by step list of directions** that need to be followed to solve a problem.
- Algorithms are often used to describe **how a computer might solve a problem**.
- A recipe can be a type of algorithm.

  It tells what ingredients are needed to make the dish and what steps to follow. If the recipe tells exactly what to do without too much confusion, then it is an algorithm.

## How does this algorithm work?

Given is an array $A$ of size $n$ of integer numbers.

```
for ( i = 1 ; i < n ; i++ )
{
        int swaps = 0;
        for ( j = 0 ; j < n – i; j++ )
        {
                if ( a[j] > a[j + 1] )
                {
                        swap( a[j], a[j + 1] );
                        swaps = swaps + 1;
                }
        }
        if ( swaps == 0 )  then  break;
}
```

## Bubble Sort

Given is an array $A$ of size $n$ of integer numbers.

```
for ( i = 1 ; i < n ; i++ )
{
        int swaps = 0;
        for ( j = 0 ; j < n – i; j++ )
        {
                if ( a[j] > a[j + 1] )
                {
                        swap( a[j], a[j + 1] );
                        swaps = swaps + 1;
                }
        }
        if ( swaps == 0 )  then  break;
}
```

## Bubble Sort

```
1.    [5, 2, 7, 3, 9, 11]
2.    [2, 5, 7, 3, 9, 11]
3.    [2, 5, 3, 7, 9, 11]   iteration 1
4.    [2, 5, 3, 7, 9, 11]
5.    [2, 3, 5, 7, 9, 11]
                            iteration 2
6.    [2, 3, 5, 7, 9, 11]
7.    [2, 3, 5, 7, 9, 11]   iteration 3
```

## Bubble Sort

- It is impractical sorting algorithm as other sorting algorithms of the same complexity generally run faster.
- The good thing is that it quickly detects that the list is already sorted. It does this faster than some other (even better) algorithms.

  However, it is not alone in doing this.

## How does this algorithm work?

Given is an array $A$ of size $n$ of integer numbers.

```
for ( i = 1 ; i < n ; i++ )
{
        j = i;
        while ( ( j > 0 ) && ( A[j-1] > A[j] ) )
        {
                swap( A[j], A[j-1] );
                j = j – 1;
        }
}
```

## Insertion Sort

Given is an array $A$ of size $n$ of integer numbers.

```
for ( i = 1 ; i < n ; i++ )
{
        j = i;
        while ( ( j > 0 ) && ( A[j-1] > A[j] ) )
        {
                swap( A[j], A[j-1] );
                j = j – 1;
        }
}
```

## Insertion Sort

```
1.     [5, 2, 7, 3, 9, 11]
2.     [5, 2, 7, 3, 9, 11]
3.     [5, 2, 7, 3, 9, 11]
4.     [2, 5, 7, 3, 9, 11]
5.     [2, 5, 7, 3, 9, 11]
6.     [2, 5, 7, 3, 9, 11]
7.     [2, 5, 3, 7, 9, 11]
8.     [2, 3, 5, 7, 9, 11]
9.     [2, 3, 5, 7, 9, 11]
10.    [2, 3, 5, 7, 9, 11]
11.    [2, 3, 5, 7, 9, 11]
```

## Insertion Sort

- **Adaptive**, if Insertion Sort is run on a nearly sorted list, it has to do far **less work**.

  The way the data is presented makes a difference!

- **Stable**; i.e., does not change the relative order of elements with equal keys.

- **In-place**; i.e., requires only a constant number of elements of additional memory.

## How does this algorithm work?

Given is an array $A$ of size $n$ of integer numbers.

```
for ( int i = 0 ; i < n; i++ )
{
        int min = i;
        for ( int j = i+1; j < n; j++ )
        {
                if ( A[j] < A[min] )  then  min = j;
        }
        if ( min != i )  then  swap( A[i], A[min] );
}
```

## Selection Sort

Given is an array $A$ of size $n$ of integer numbers.

```
for ( int i = 0 ; i < n; i++ )
{
        int min = i;
        for ( int j = i+1; j < n; j++ )
        {
                if ( A[j] < A[min] )  then  min = j;
        }
        if ( min != i )  then  swap( A[i], A[min] );
}
```

## Selection Sort

```
1.    [5, 2, 7, 3, 9, 11]
2.    [5, 2, 7, 3, 9, 11]
3.    [2, 5, 7, 3, 9, 11]
4.    [2, 5, 7, 3, 9, 11]
5.    [2, 3, 7, 5, 9, 11]
6.    [2, 3, 7, 5, 9, 11]
7.    [2, 3, 5, 7, 9, 11]
8.    [2, 3, 5, 7, 9, 11]
9.    [2, 3, 5, 7, 9, 11]
10.   [2, 3, 5, 7, 9, 11]
```

## Selection Sort

- Selection Sort generally does the same number of comparisons, however there is no “drop out” like there is in Insertion Sort.
- This makes it much **less efficient**.

## Running Time $T(n)$

The **running time** of a given algorithm is **the number of elementary operations** to reach a solution, for example

- Sorting – number of array elements
- Arithmetic operation – number of bits
- Graph search – number of vertices and edges

We can evaluate running time by

- Operation Counting
- Asymptotic Notations
- Substitution Method
- Recurrence Tree
- Master Method

## Operation Counting

```
int LinearSearch( int[] A, int n, int value )
{
    for ( int i = 0 ; i < n; i++ )      1 (initialization) + n+1 (checks) operations
    {
        if ( A[i] == value )            n (checks) operations
            then return i;              1 (return) operation
    }                                   n (increment) operations
    return -1;                          1 (return) operation
}
```

- Worst case: $T(n) = 1 + (n+1) + n + 1 + n = 3n + 3$ operations
- Best case: $T(n) = 1 + 1 + 1 + 1 = 4$ operations

## Best, Worst and Average Case Complexity

Running the same algorithm on different inputs may yield different running times.

Back to the Linear Search algorithm

- **Worst case running time:** What if the value does not belong to the data array?
- **Best case running time:** What if the first element matches the value to search for?
- **Average case running time:** What if the value exists somewhere in the middle of the data array?

## Focus on Worst Case

When comparing two algorithms, consider the worst-case running time complexity. Why?

- It provides an upper bound on the runtime.
- It happens fairly often.
- Average case usually is similar to the worst case.

## Operation Counting: Dealing with nested loops

```
InsertionSort( int[] A, int n )
for ( int j = 1 ; j < n; j++ )              1 (initialization) + n (checks) operations
{
    int temp = A[j];                        n-1 (assignment) operation
    int i = j −1;                           n-1 (assignment) operation
    while ( ( i ≥ 0 ) && ( A[i] > temp ) )  $2\sum_{j=1}^{n} j$-1 (checks) operations
    {
        A[i + 1] = A[i];                    $\sum_{j=1}^{n-1} j$ (assignment) operations
        i = i−1;                            $\sum_{j=1}^{n-1} j$ (decrement) operations
    }
    A[i + 1] = temp;                        n-1 (assignment) operations
}                                           n-1 (incremenet) operations
```

Worst case: $T(n) = 1 + n + (n-1) + (n-1) + 2(n(n+1)/2 - 1) +$
$2n(n-1)/2 + (n-1) + (n-1)$
$= 2n^2 + 5n - 5$ operations

Arithmetic series: $\sum_{j=1}^{n} j = \frac{1}{2} n(n+1)$

## Operation Counting: Dealing with nested loops

```
InsertionSort( int[] A, int n )
for ( int j = 1 ; j < n; j++ )                 1 (initialization) + n (checks) operations
{
    int temp = A[j];                           n-1 (assignment) operation
    int i = j −1;                              n-1 (assignment) operation
    while ( ( i ≥ 0 ) && ( A[i] > temp ) )     2 $\sum_{j=1}^{n} j$-1 (checks) operations
    {
        A[i + 1] = A[i];                       $\sum_{j=1}^{n-1} j$ (assignment) operations
        i = i−1;                               $\sum_{j=1}^{n-1} j$ (decrement) operations
    }
    A[i + 1] = temp;                           n-1 (assignment) operations
}                                              n-1 (incremenet) operations
```

Best case: $T(n) = 1 + n + (n-1) + (n-1) + 2(n-1) + (n-1)$

$= 6n - 4$ operations

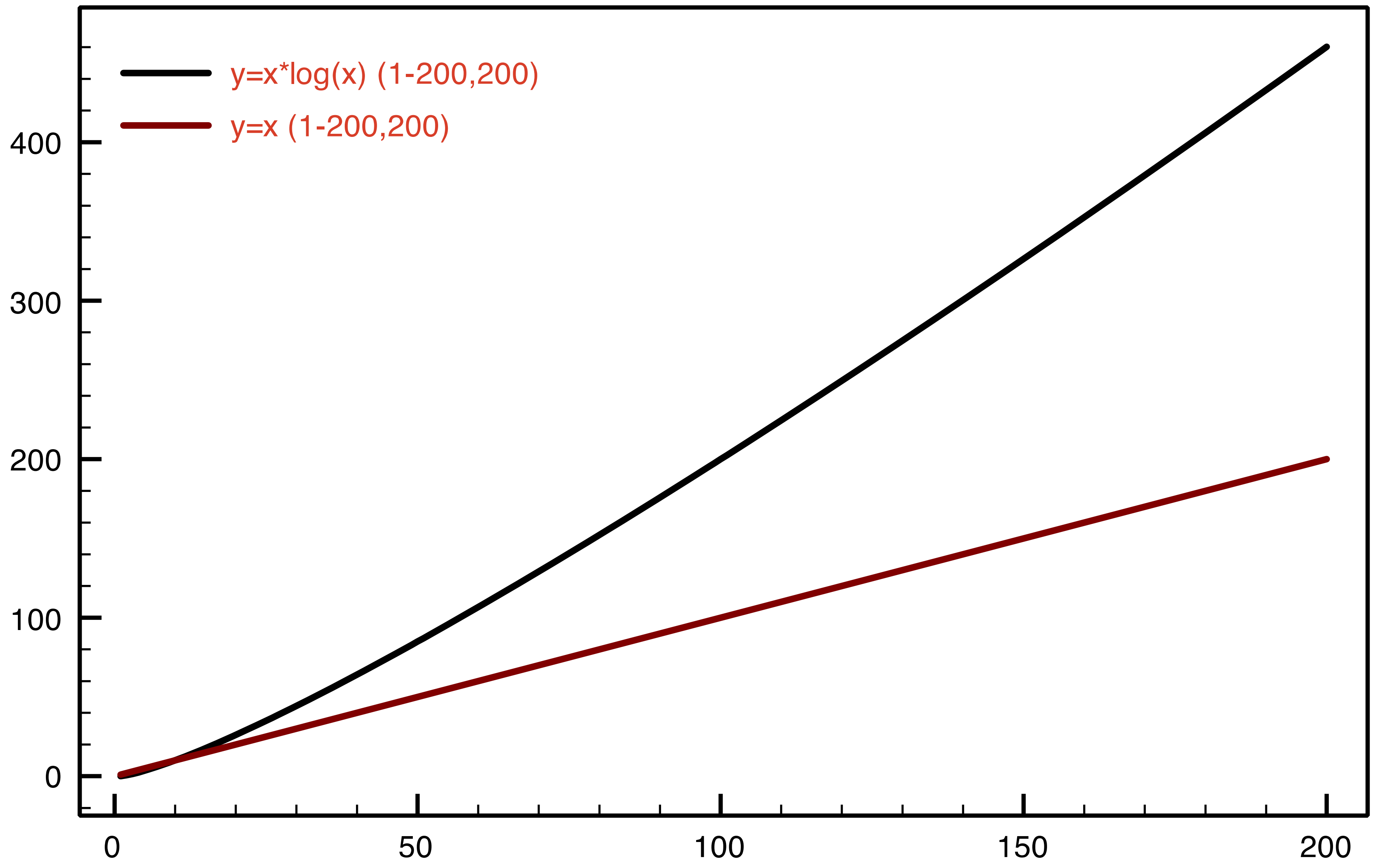
## Operation Counting: General rules

- **Sequence:** Add the running times of consecutive statements.
- **Loops:** The running time of the inner loop statements times the product of the sizes of all loops.
- **Control Statement:** The maximum of running times of case 1, case 2 , … case k.

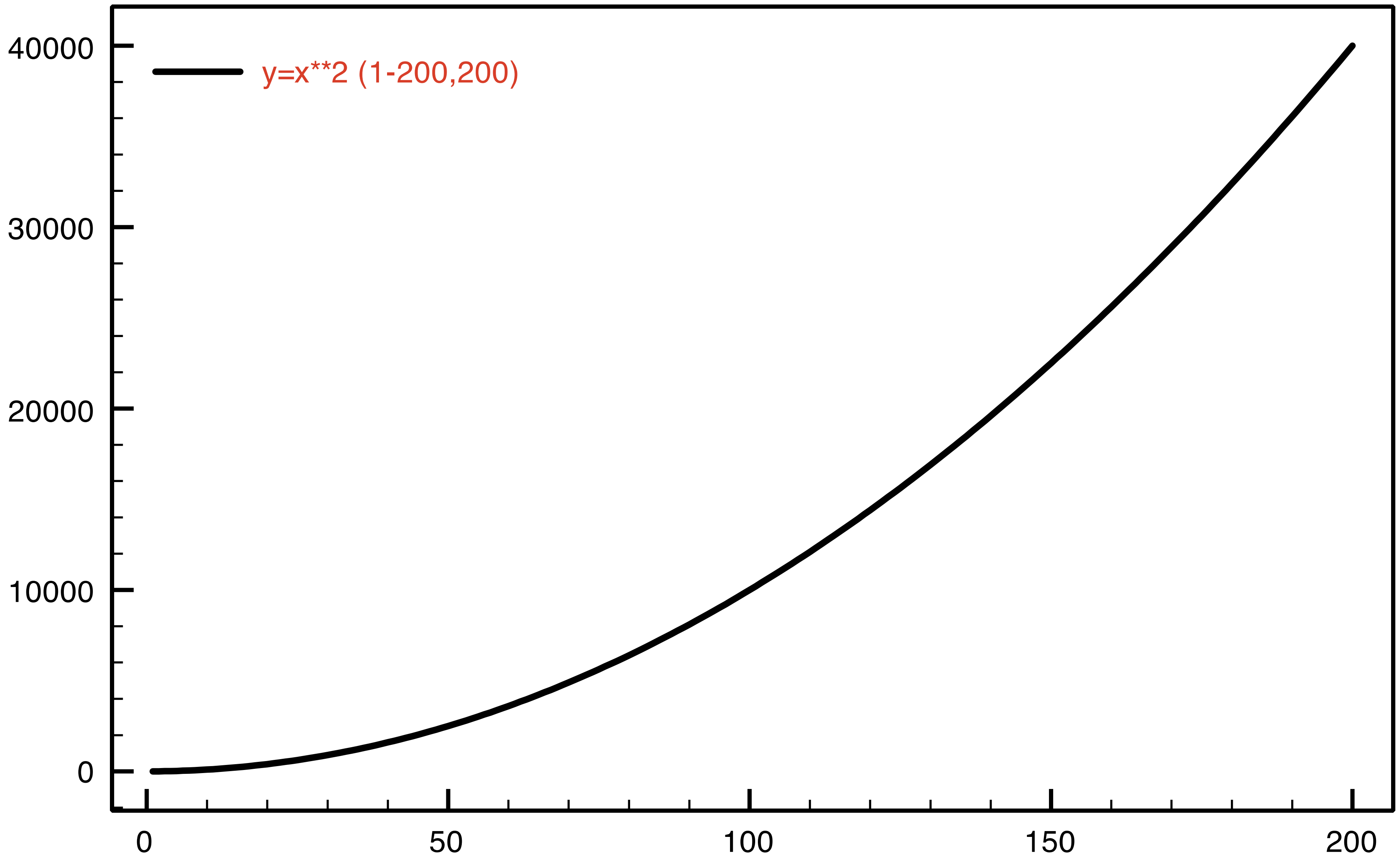
## Efficiency: What's our measure?

- Let $n$ be the number of input elements (for example, the number of integers in the array).
- Measure time for operations on data structures and time to execute algorithms in dependence (function) of $n$.
- Consider orders of magnitude.

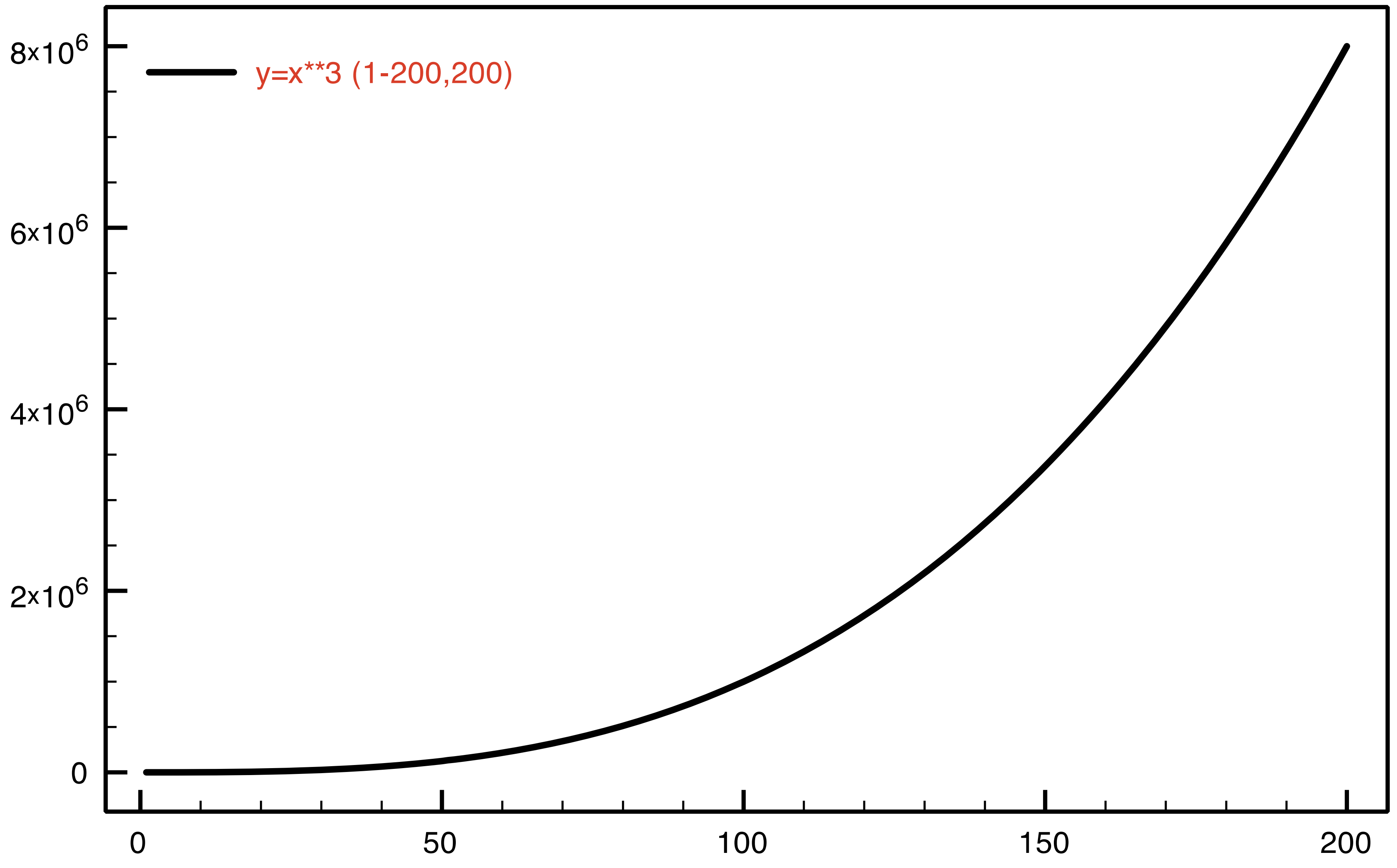
## (Almost) Linear Runtime



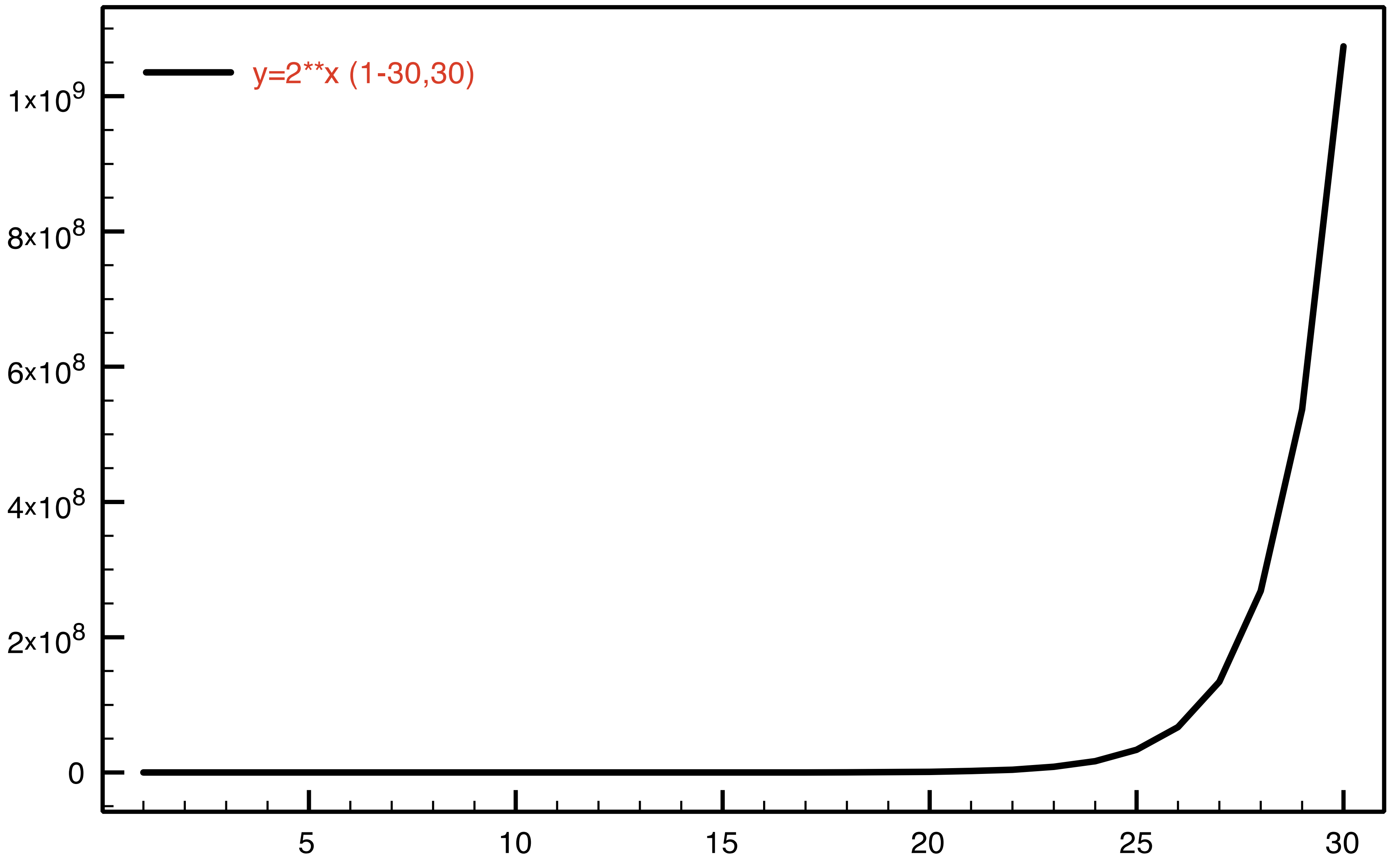
## Quadratic Runtime



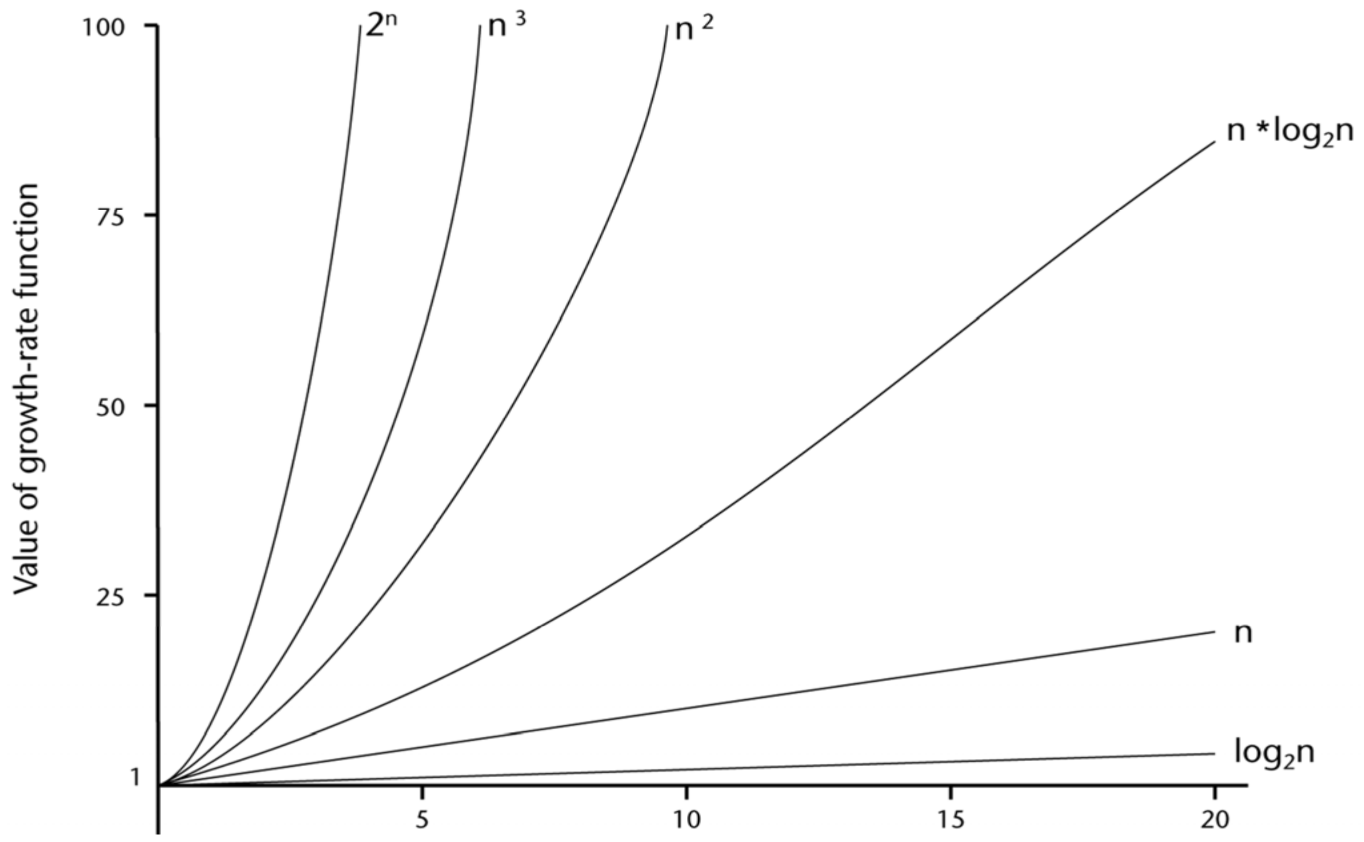
## Cubic Runtime



## Exponential Runtime



## Order of Growth



- Lower-order terms are not significant for large $n$
- Constants and coefficients are less significant

## Complexity

| n | n $\log_{10}$ n | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|
| 10 | 10 | 100 | 1000 | 1.024 |
| 100 | 200 | 10.000 | 1.000.000 | 2^100 |
| 1000 | 3.000 | 1.000.000 | 1.000.000.000 | 2^1000 |
| 10000 | 40.000 | 100.000.000 | $10^{12}$ | 2^10000 |

- It’s great to have algorithms that run in linear time or time $n \log n$.
- Many important problems have algorithms whose runtime is bounded by a small polynomial (e. g. $n^2$ or $n^3$)
- For a wide class of important problems there is most probably no algorithm that runs in polynomial time.

## Machines do not really dominate the efficiency

Sort an array of $n = 1000000$ integers with two algorithms:

- Algorithm A has $2n^2$ complexity
- Algorithm B has $50n \times log(n)$ complexity

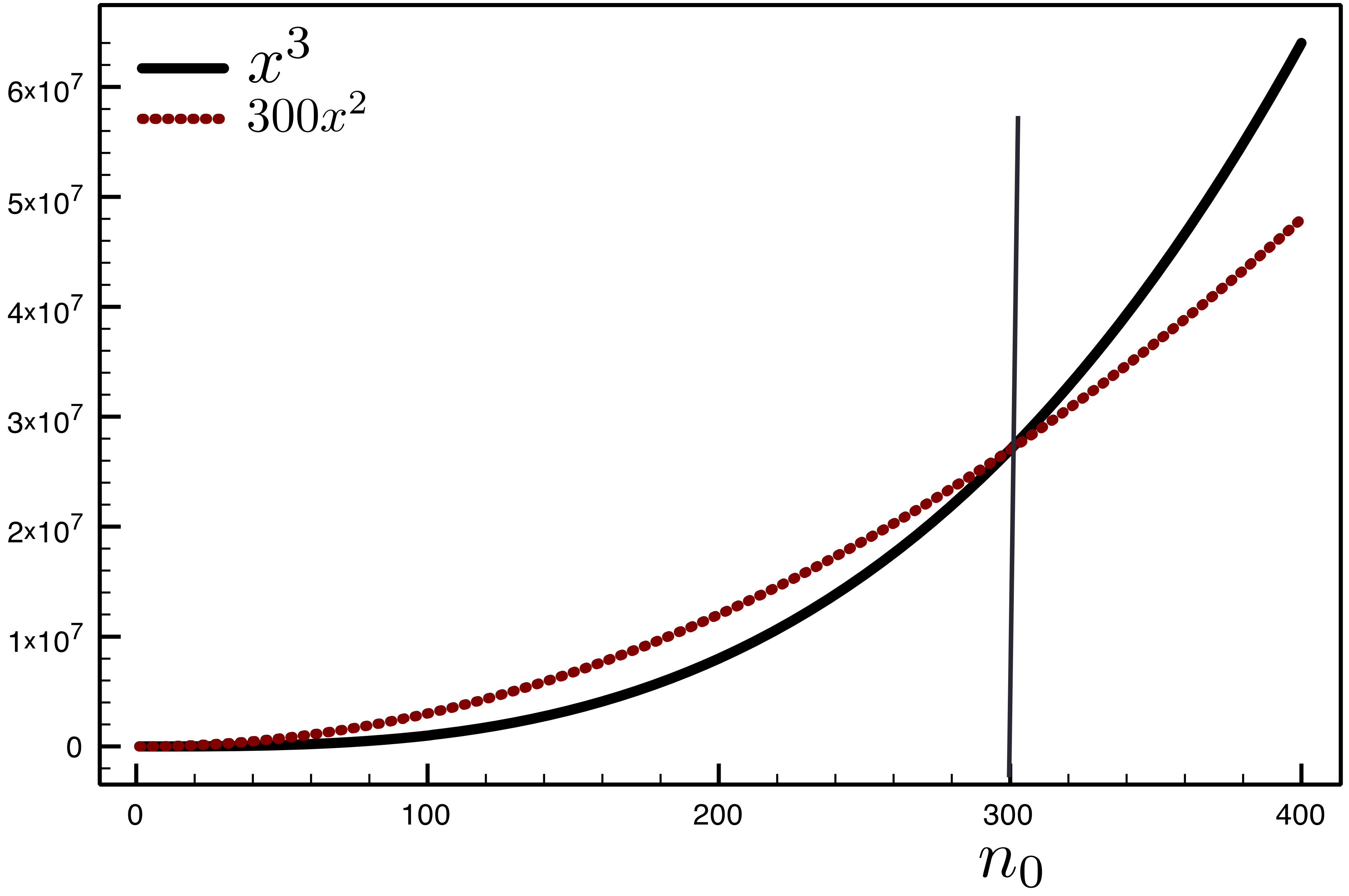
Given are two machines:

- PC1 ($10^9$ instructions per second), runs algorithm A $\Rightarrow$ runtime of 2 $\times \frac{(10^6)^2}{10^9} = 2000\ sec$
- PC2 ($10^7$ instructions per second), runs algorithm B $\Rightarrow$ runtime of 50 $\times \frac{(10^6) \times log(10^6)}{10^7} \approx 100\ sec$

## Asymptotic behavior

- We measure runtime as a function of the input size $n$.
- Define complexity depending on the asymptotic behavior.
- Want to have algorithms that solve a given problem and have low complexity.

## Asymptotic behavior



## Asymptotic notations

We want to measure computation times **asymptotically** and describe them as the relative growth rates of functions.

$$O(g(n)) = \{\varphi(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : \varphi(n) \leq cg(n)\}$$
$$\Omega(g(n)) = \{\varphi(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : \varphi(n) \geq cg(n)\}$$
$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$
$$o(g(n)) = \{\varphi(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : \varphi(n) \leq cg(n)\}$$
$$\omega(g(n)) = \{\varphi(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : \varphi(n) \geq cg(n)\}$$

We often write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$

## Asymptotic notations

$$O\big(g(n)\big) = \{\varphi(n): \exists c > 0: \exists n_0 \in \mathbb{N}_+: \forall n \geq n_0: \varphi(n) \leq cg(n)\}$$

$$\Omega\big(g(n)\big) = \{\varphi(n): \exists c > 0: \exists n_0 \in \mathbb{N}_+: \forall n \geq n_0: \varphi(n) \geq cg(n)\}$$

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

$$o\big(g(n)\big) = \{\varphi(n): \forall c > 0: \exists n_0 \in \mathbb{N}_+: \forall n \geq n_0: \varphi(n) \leq cg(n)\}$$

$$\omega\big(g(n)\big) = \{\varphi(n): \forall c > 0: \exists n_0 \in \mathbb{N}_+: \forall n \geq n_0: \varphi(n) \geq cg(n)\}$$

– $O\big(g(n)\big)$: asymptotic upper bound (Big-O)
  A set of all functions $\varphi(n)$ such that there exist positive constants $c$ and $n_0$
  $\varphi(n) \leq cg(n)$ for all $n \geq n_0$

– $\Omega\big(g(n)\big)$: asymptotic lower bound (Big-Omega)
  A set of all functions $\varphi(n)$ such that there exist positive constants $c$ and $n_0$
  $\varphi(n) \geq cg(n)$ for all $n \geq n_0$

– $\Theta\big(g(n)\big)$: asymptotically tight bound, the same order of magnitude (Big-Theta)
  A set of all functions $\varphi(n)$ such that there exist positive constants $c_1$, $c_2$, and $n_0$
  $c_1 g(n) \leq \varphi(n) \leq c_2 g(n)$ for all $n \geq n_0$

## Asymptotic notations

### In simple words…

| | |
|---|---|
| $f(n) = O(g(n))$ | $f(n)$ grows no faster than $g(n)$ |
| $f(n) = o(g(n))$ | $f(n)$ grows slower than $g(n)$ |
| $f(n) = \Omega(g(n))$ | $f(n)$ grows at least as fast as $g(n)$ |
| $f(n) = \omega(g(n))$ | $f(n)$ grows faster than $g(n)$ |
| $f(n) = \Theta(g(n))$ | $f(n)$ grows at the same rate as $g(n)$ |

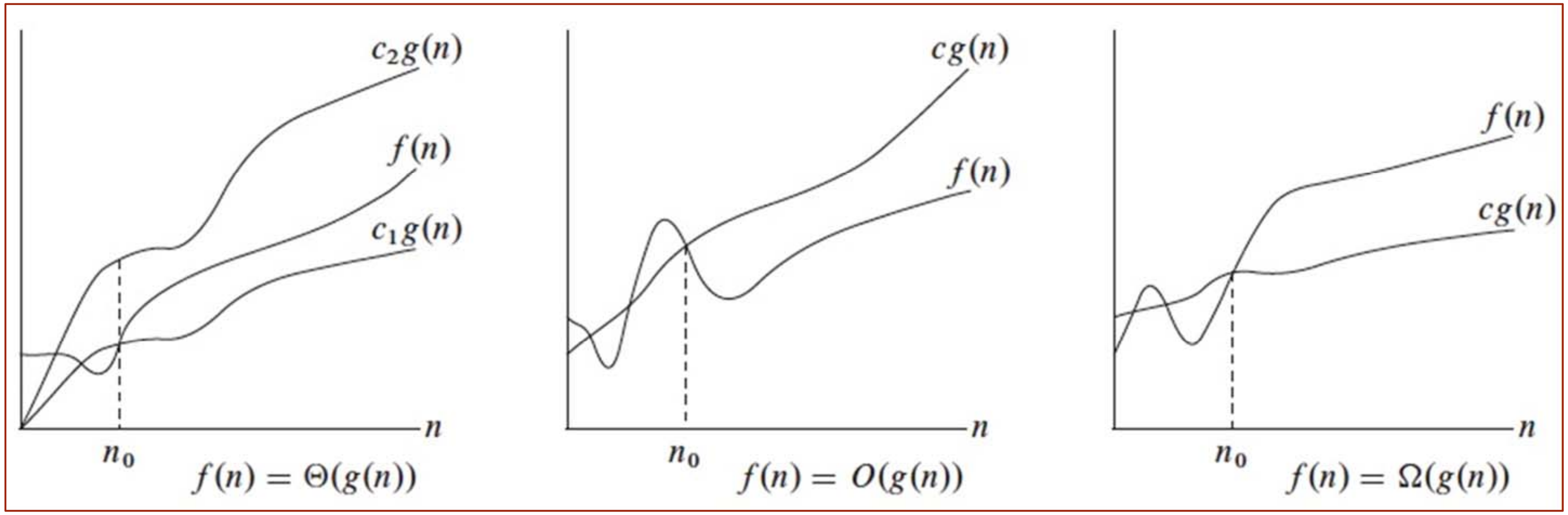
### Therefore

| | | | |
|---|---|---|---|
| $f(n) = O(g(n))$ | is equivalent to | $g(n) = \Omega(f(n))$ | and vice versa |
| $f(n) = o(g(n))$ | is equivalent to | $g(n) = \omega(f(n))$ | and vice versa |

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

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

## Asymptotic notations



The running time of an algorithm is $\Theta\big(g(n)\big)$ if and only if its worst-case running time is $O\big(g(n)\big)$ and its best-case running time is $\Omega\big(g(n)\big)$

## Asymptotic notations: Common Categories

From fastest to slowest

| | |
|---|---|
| $O(1)$ | constant (or $O(k)$ for constant $k$) |
| $O(\log n)$ | logarithmic |
| $O(n)$ | linear |
| $O(n \log n)$ | "n log n" |
| $O(n^2)$ | quadratic |
| $O(n^3)$ | cubic |
| $O(n^k)$ | polynomial (where $k$ is constant) |
| $O(k^n)$ | exponential (where constant $k > 1$) |

## Examples

$$5n : O(n), \Omega(n), \Theta(n), o(n \log n), \omega(\sqrt{n})$$

$$n^2 - n \log n : O(n^2), \Omega(n^2), \Theta(n^2), o(n^3), \omega(n \log n)$$

$$100n : O(n^2), \Omega(\sqrt{n}), \Theta(n), o(n \log n), \omega(\sqrt{n})$$

## Right or Wrong

| | |
|---|---|
| $5n\log n \in O(n\log n)$ | Right |
| $5n\log n \in O(n^2)$ | Right |
| $5n\log n \in \Omega(n^2)$ | Wrong |
| $5n\log n \in o(n^2)$ | Right |
| $5n\log n + n^2 \in O(n\log n)$ | Wrong |
| $5n\log n + n^2 \in O(n^2)$ | Right |

## The Limit Rule

Suppose the ratio’s limit

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$$

exists (may be infinite, ∞), then

- $f(n) \in O\big(g(n)\big)$ if $L \neq \infty$
- $f(n) \in o\big(g(n)\big)$ if $L = 0$
- $f(n) \in \Omega\big(g(n)\big)$ if $L \neq 0$
- $f(n) \in \omega\big(g(n)\big)$ if $L = \infty$
- $f(n) \in \Theta\big(g(n)\big)$ if $0 < L < \infty$

## The Limit Rule: Applying the L'Hopital Rule

When $f$ and $g$ are positive and differentiable functions for $x > 0$, but $\lim_{x\to\infty} f(x) = \lim_{x\to\infty} g(x) = \infty$ or $\lim_{x\to\infty} f(x) = \lim_{x\to\infty} g(x) = 0$, the limit $L$ can be computed using the standard **L'Hopital** rule of calculus:

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

where $z'(x)$ denotes the first derivative of the function $z(x)$.

## General rules and relations

- **Scaling:**
  for all constant factors $c > 0$ the function $cf(n) = O(f(n))$

- **Transitivity:**
  if $f(n) = O(g(n))$ and $g(n) = O\big(h(n)\big)$,
  then $f(n) = O(h(n))$

- **Rule of Sums:**
  if $f_1(n) = O(g_1(n))$ and $f_2(n) = O\big(g_2(n)\big)$,
  then $f_1(n) + f_2(n) = O(max\{g_1(n), g_2(n)\})$

- **Rule of Products:**
  if $f_1(n) = O(g_1(n))$ and $f_2(n) = O\big(g_2(n)\big)$,
  then $f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n))$

## Summary

- Efficient data structures and algorithms are crucial for successful computer applications.
- Measure runtime as a function of the given input size.
- Examine asymptotic behavior and refer to complexity classes.

## Other references and things to do

- Watch the attached Youtube video in CloudDeakin.
- Read chapters 4.2, 4.3 in Data Structures and Algorithms in Java. Michael T. Goodrich, Irvine Roberto Tamassia, and Michael H. Goldwasser, 2014.
- Study the attached “The Asymptotic Cheat Sheet” in CloudDeakin.