# Algorithm Analysis

- Performance Relations
  - **Time and Space**
- Classification of algorithms
- Comparison between classes
- Statistical cases – **best, average, worst**
- Example analysis
  - Bubble sort
  - Binary search

# **Time & Space** Performance estimates

- **Time performance**
  - the relationship between the **size of the data** collection and **time to process**
  - Example: sort an array of n elements –

- **Space performance**
  - the relationship between the **memory required** to solve a problem and the **problem size**
  - Example: sort a list of integers with 2 stacks – the extra stack space represents an overhead.

# Performance

- T and S are described in terms of the size of the problem **(number of elements)**
  - **Time** = **T(n)**
    - time to solve a problem with **input size n**
  - **Space** = **S(n)**
    - space to solve a problem with **input size n**

# Performance

- The **relationship** between the **input size** and **time/space**
- If n is the size of the input
  - Ex: $T(n) = n^2 + 3n$
  - Ex: $S(n) = \log(n)$

# Performance

- To **compare** the performance of algorithms we use T(n) and S(n)
- These are written in the form: **O(n)**
- O is pronounced (**Big-Oh**) - O is the Order of growth
- O determines a performance class
- **Comparing algorithms** on an abstract level uses

  **Big-O** as an **INDICATOR** of the performance

# O(n) – how is T(n) used?

- Rule of thumb: for $T(n) = 3n^4 + n^2$ the class is $O(n^4)$
  - For large values of n, $n^4$ is more significant than $n^2$
- Mathematically for $T(n) = 3n^4 + n^2$ there exists a class $O(cn^4)$ where $cn^4 >= T(n)$ for some c
  - For $T(n) = n^2 + 2n + 1$, for what c is $cn^2 >= T(n)$?
  - $c = 4, n = 1$, therefore the class is $O(4n^2)$
- **Constants in O-notation disappear, $O(5n^2 + 3) \equiv O(n^2)$**
  - The reason: the constant does not depend on n
  - Since O denotes a class T(n) belongs to some class O, $T(n) = n^2 + 2n + 1 \in O(n^2)$

# O(n) (continued)

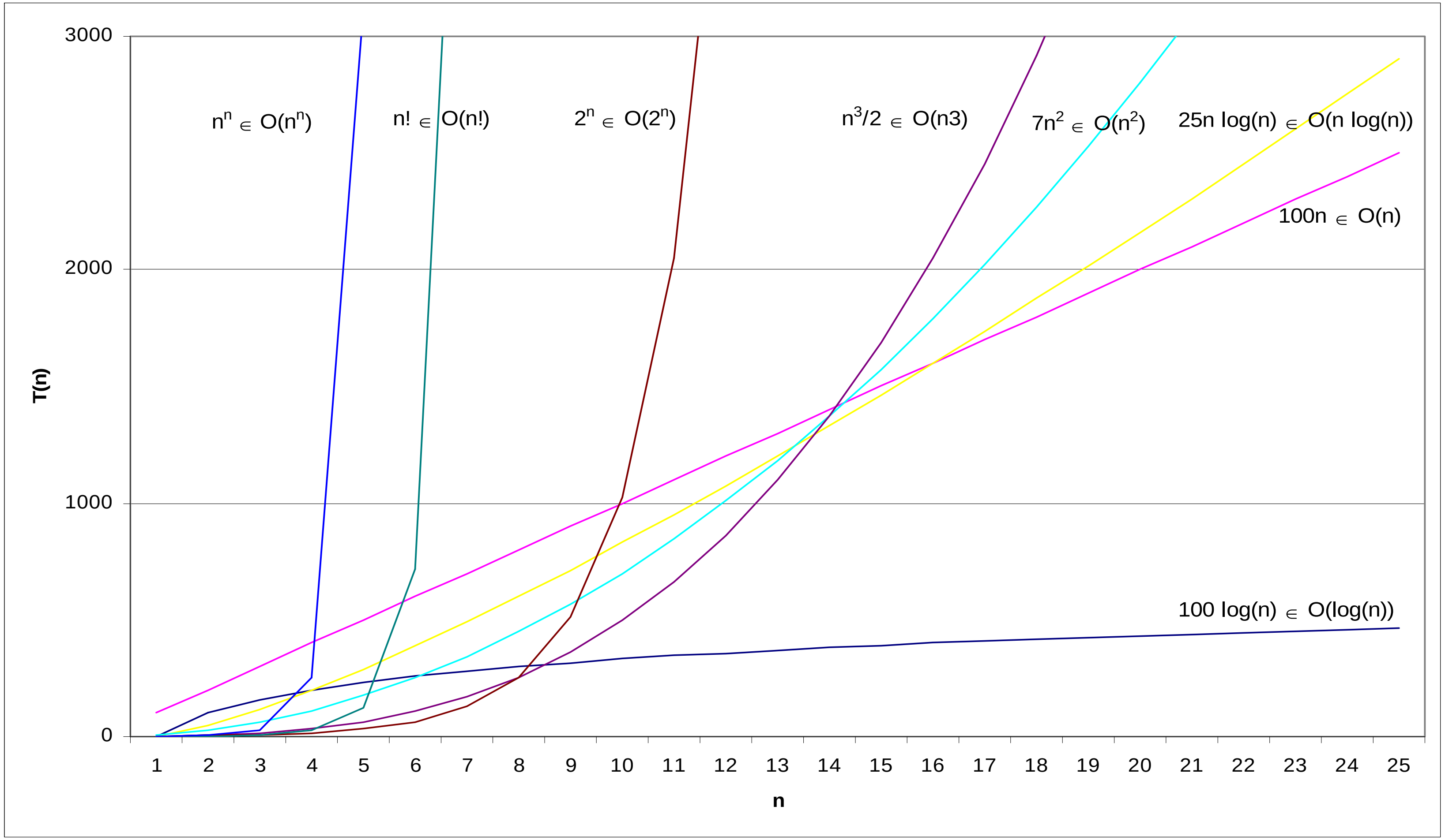
- How many different classes are there??
- Infinitely many… BUT only a few are of interest

**$O(1)$, $O(\log(n))$, $O(n)$, $O(n \log(n))$, $O(n^2)$, $O(n^3)$,**

…, **$O(x^n)$**, …, **$O(n!)$**, …, **$O(n^n)$** …

- The **performance** is given in **decreasing order**

# Diagram



3000
2000
1000
0
T(n)
n^n ∈ O(n^n)
n! ∈ O(n!)
2^n ∈ O(2^n)
n^3/2 ∈ O(n3)
7n^2 ∈ O(n^2)
25n log(n) ∈ O(n log(n))
100n ∈ O(n)
100 log(n) ∈ O(log(n))
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
n

# From the diagram

- Certain problems do not fall into these classes
  - Ex: **$7n^2$** is **better** than **100n** for problem with **size < 14**
  - This is despite the fact that **$O(n^2)$** is worse than **O(n)**
- Assume that **n is quite large:-** therefore **big-O** is significant
- Implementing an algorithm which is **O(n!)** is not a good idea!

# Sorting has different Big-Oh solutions

1. To sort a sequence with 2 elements is **O(1)**
   - requires max 2 operations:
   - compare the elements and swap.
2. To sort an already sorted sequence in reverse order requires **O($n^2$)**
3. To sort a random sequence takes **O(n log(n))** **(quicksort)**

- These are called **best, worst** and **average** cases
- In general, the average case is of most interest as long as the worst case does not occur often

# Analysis of bubble sort

```
bubble(A)
  for i = 1 upto A.size - 1
     for j = A.size downto i+1
        if A[j-1] > A[j]
        then swap(A[j-1], A[j])
     end for
  end for
end bubble
```

- n = array size
- Outer loop executes n times
- Inner  loop executes x times
  - x är n - i

$$\sum_{i=1}^{n-1}\sum_{j=n}^{n-i} ifsats \,\&\, swap$$

# Bubble sort (continued)

- The inner loop executes **n-1** times
- **i** is decided by the outer loop
- The if and swap execute **(n - 1) + (n - 2) + (n - 3) + … + (n - n)** times
- If you remember arithmetic series then **(n - 1) * n / 2** times
- This gives **$(n^2 - n) / 2$** ➔ **$O(n^2)$**

$$\sum_{i=1}^{n-1} \sum_{j=n}^{n-i} ifsats\ \&\ swap$$

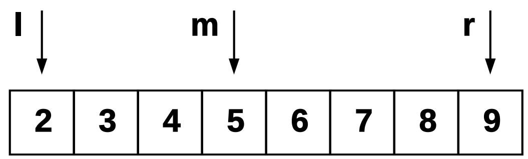
$$T(n) = \frac{n^2 - n}{2} \in O(n^2)$$

# Example analysis (continued)

- **Binary search analysis**
- n = array size
- The first time we are looking for 1 element in n
- The next time 1 element in n / 2
- The next time n / 2 / 2 element

  …
- …
- Finally there is one element left in the search space
- **How many times did we divide the search space?**

```
binsearch(A, v, l, r)
   let  m = (l + r) / 2
   if A[m] == v then return m
   else if l==r then return NOTFOUND
   else if v < A[m]
   then return binsearch(A,v,l,m-1)
   else return binsearch(A,v,m+1,r)
end binsearch
```



# Example analysis (continued)

- **In other words how many times do we have to partition the search space before we have 1 element (worst case)?**
- Suppose that the **number of partitions is k** and **array size n**
- Then:

$$\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n \Rightarrow k = c \cdot \log n$$

- since k is the number of operations required to solve a problem with input size n, therefore T(n) = k
- **if T(n) = c log(n), then T(n) $\in$ O(log(n))**

# Rules of Thumb for Analysis

- Ex 1: **Nested loops**. Big-O is
  - **$O(n^{\text{number of loops}})$ if the loop depends on N**
- Ex 2: "**divide and conquer**" algorithm.
  - Such algorithms are logarithmic.
  - A decision is made which divides the problem space, the choice eliminates a part of the problem.
  - The rule is **$O(\log(n))$**

# Tips

- **Do not analyse algorithms** – waste of time! (wot ☺)
- **Do not write your own algorithms** to solve known problems
- **Use already proven and certified algorithms**

- If you have to choose between well-known algorithms
  - Choose on the basis of problem size (n)
  - If (n) is of arbitrary size
    - choose the algorithm with the best Big-O classification
  - If the algorithms have the same Big-O classification
    - Consider T(n) and/or S(n) performance
    - **Consider the best / worst / average case**