

---

# Estruturas de Informação

## Algorithm Analysis

Fátima Rodrigues

[mfc@isep.ipp.pt](mailto:mfc@isep.ipp.pt)

Departamento de Engenharia Informática (DEI/ISEP)

# Estruturas de Informação

---

This subject is about the design of “good” data structures and algorithms

A ***data structure*** is a systematic way of organizing and accessing data

An ***algorithm*** is a step-by-step procedure for performing some task in a finite amount of time

# What makes a good algorithm?

---

For different algorithms that solve the same problem, an algorithm is more efficient, if it need less resources to solve the same problem

Complexity refers to the rate at which the storage or time grows as a function of the input to the algorithm

- **$T(n)$  = time complexity:** amount of time an algorithm will take based on input
- **$S(n)$  = space complexity:** amount of memory space an algorithm will take based on input

# Algorithm Analysis

---

We want a method for determining the relative speed of algorithms that:

- doesn't depend upon hardware used (e.g., PC, Mac, etc.)
- the clock speed of your processor
- what compiler you use
- even what language you write in

# Algorithm Analysis

---

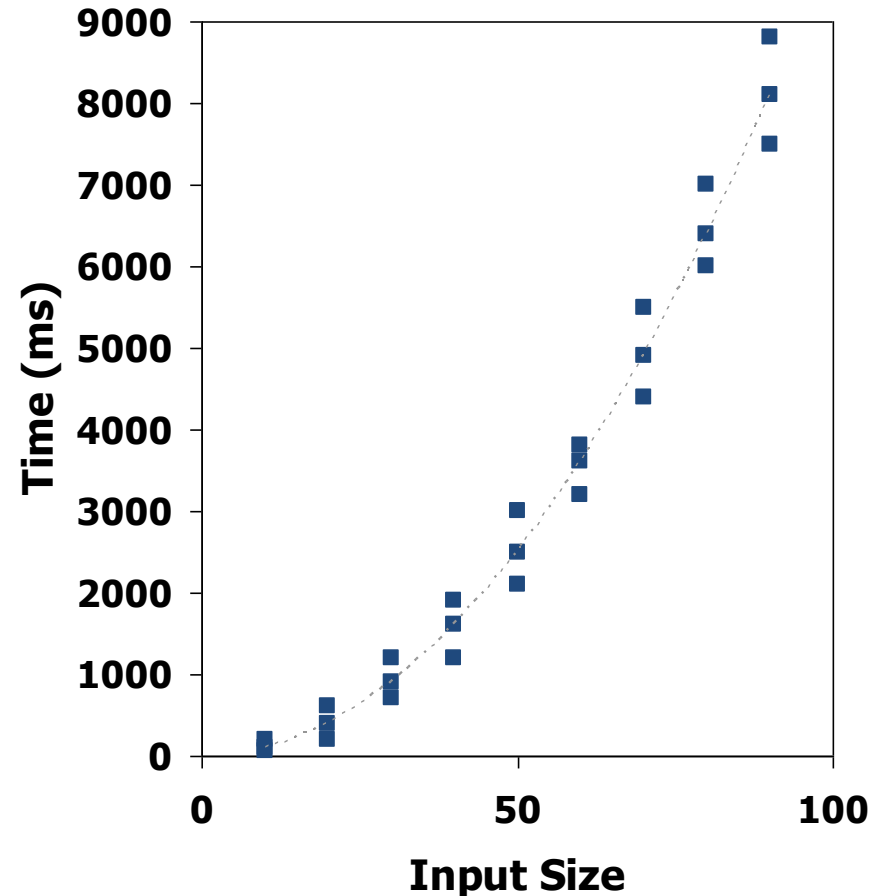
A machine-independent way to describe execution time

The purpose of Algorithm Analysis is:

to describe change in execution time relative to change in input size, in a way that is independent of issues such as machine times or compilers

# Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like `clock()` to get an accurate measure of the actual running time
- Plot the results



# Limitations of Experiments

---

- It is necessary to fully implement the algorithm, which may be time expensive
- In order to compare two algorithms, the same hardware and software environments must be used
- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment

# Theoretical Analysis

---

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size,  $n$
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment



# Algorithm Analysis

---

- Suppose that algorithm  $A$  processes  $n$  data elements in time  $T$
- Algorithm analysis attempts to estimate how  $T$  is affected by changes in  $n$ .
- In other words,  $T$  is a function of  $n$  when we use  $A$

# A Simple Example

---

- Suppose we have an algorithm that is  $O(n)$   
E.g., summing elements of array
- Suppose to sum 10,000 elements takes 32 ms
- How long to sum 20,000 elements?

# Non-Linear Times

---

- Suppose we have an algorithm that is  $O(n^2)$   
E.g., summing elements of a matrix
- Suppose input size  $n$  doubles, what happens to execution time?

It goes up by 4

Why 4?

Need to figure out how to do this ...

# The Calculation

---

The ratio of the big-Oh sizes should equal the ratio of the execution times

$$\frac{n_1^2}{n_2^2} = \frac{t_1}{t_2}$$

We increased  $n$  by a factor of two:

$$\frac{n^2}{(2n)^2} = \frac{t}{x}$$

then solve for  $x$

# Some typical functions

---

Basically there are two types of algorithms:

- those with running time limited by a **polynomial function** dependent on the input data size - **Efficient algorithms**
- and those who typically have an **exponential** evolution - **not efficient algorithms**

Some functions that often appear in algorithm analysis

## Polynomial:

- Constant  $\approx 1$
- Logarithmic  $\approx \log n$
- Linear  $\approx n$
- N-Log-N  $\approx n \log n$
- Quadratic  $\approx n^2$
- Cubic  $\approx n^3$

## Exponential:

- $2^n$
- $3^n$
- $n!$

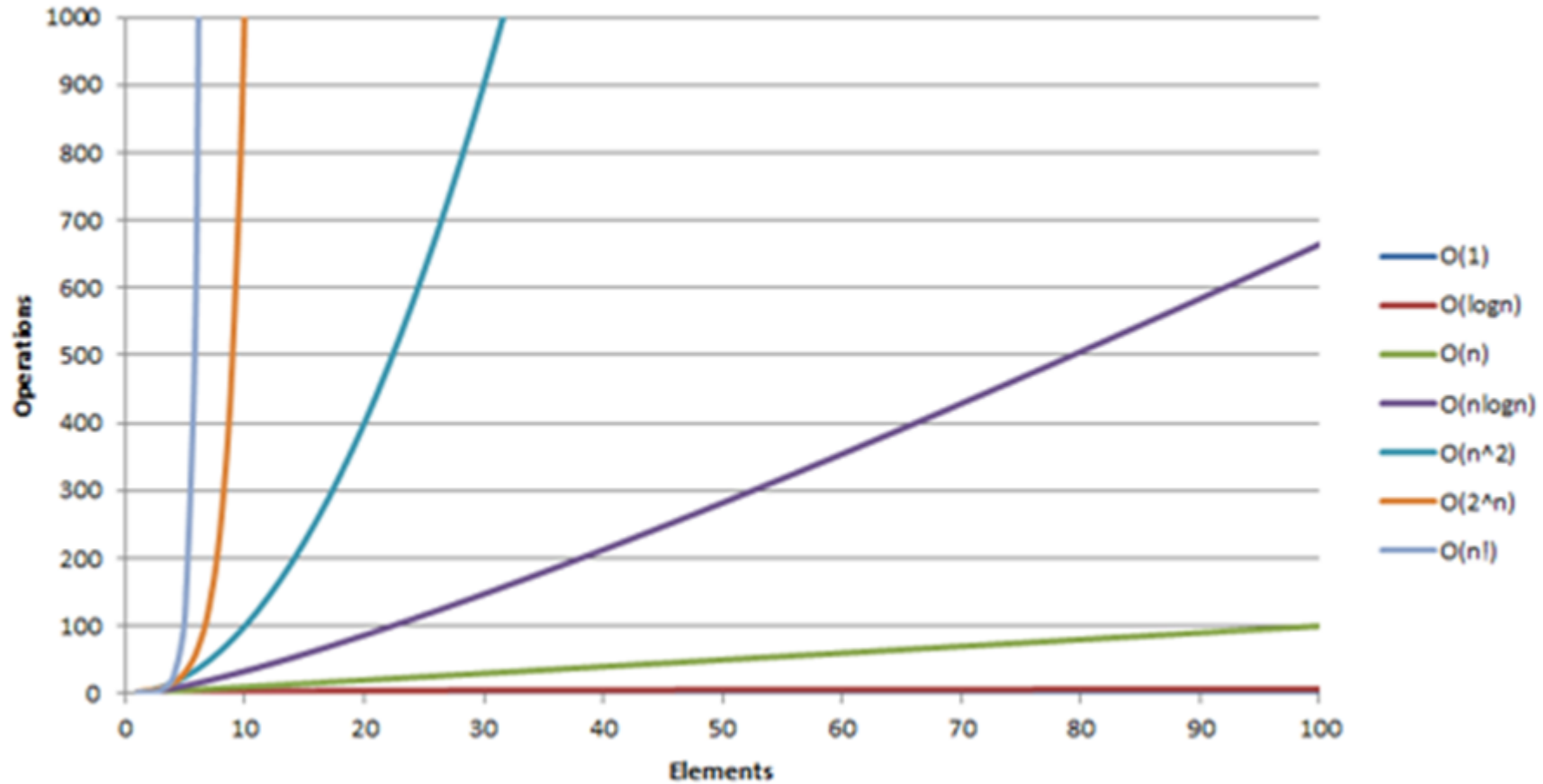
# Most common Complexity Orders

---

Input data:  $n = 10^5$  elements

Execution time of each step  $k = 10^{-5}$  seconds =  $10 \mu\text{s}$

Function	Name	Time
$n!$	Factorial	
$2^n$ (or $c^n$ )	Exponential	50 000 hours
$n^d, d > 3$	Polynomial	
$n^3$	Cubic	
$n^2$	Quadratic	28 hours
$n\sqrt{n}$		
$n \log n$		17 seconds
$n$	Linear	1 second
$\sqrt{n}$	Root-n	$3.2 \times 10^{-4}$ sec
$\log n$	Logarithmic	$170 \mu\text{s}$
1	Constant	$10 \mu\text{s}$



# Counting Primitive Operations

By inspecting the pseudocode, we can determine the maximum number of **primitive operations** executed by an algorithm, as a function of the input size

**Example:** Find the maximum value in a vector

		# Operations
1	<b>Algorithm</b> <b>int</b> arrayMax ( <b>int</b> A[], <b>int</b> n)	
2	{     currentMax $\leftarrow$ A[0];	1
3	<b>for</b> (i $\leftarrow$ 1 ; i < n ; i++)	1+n+1+n
4	<b>if</b> (A[i] > currentMax)	n
5	currentMax $\leftarrow$ A[i];	n
6	<b>return</b> currentMax ;	1
	}	4n+4



# Three different Notations $O$ , $\Omega$ e $\Theta$

---

## big-Oh

“ $T(n)$  is  $O(f(n))$ ” iff for some constants  $c$  e  $n_0$ ,  $T(n) \leq cf(n)$  for all  $n \geq n_0$

## big-Omega

“ $T(n)$  is  $\Omega(f(n))$ ” iff for some constants  $c$  e  $n_0$ ,  $T(n) \geq cf(n)$  for all  $n \geq n_0$

## big-Theta

“ $T(n)$  is  $\Theta(f(n))$ ” iff  $T(n)$  is  $O(f(n))$  and  $T(n)$  is  $\Omega(f(n))$

Informally:

$T(n)$  is  $O(f(n))$  basically means that  $f(n)$  describes the **upper bound** for  $T(n)$

$T(n)$  is  $\Omega(f(n))$  basically means that  $f(n)$  describes the **lower bound** for  $T(n)$

$T(n)$  is  $\Theta(f(n))$  basically means that  $f(n)$  describes the **exact bound** for  $T(n)$

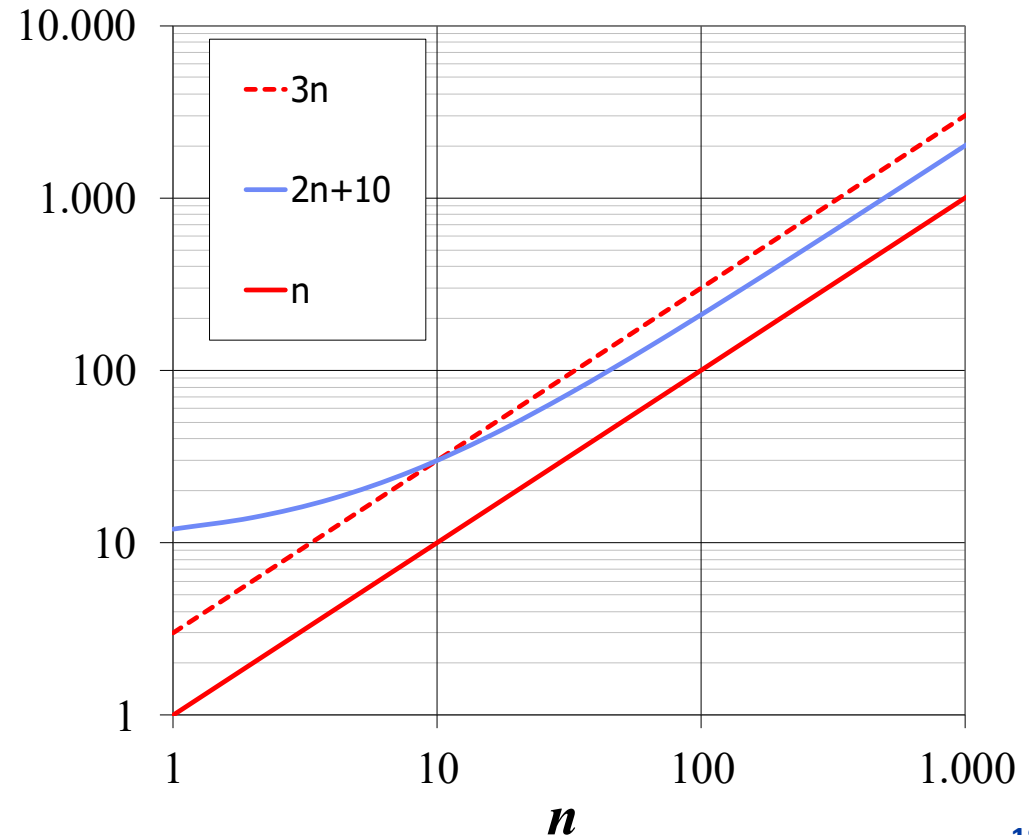
# Big-Oh Notation

Given functions  $f(n)$  and  $g(n)$ , we say that  $f(n)$  is  $O(g(n))$  if there are positive constants  $c$  and  $n_0$  such that

$$f(n) \leq cg(n) \text{ for } n \geq n_0$$

Example:  $2n + 10$  is  $O(n)$

- $2n + 10 \leq cn$
- $(c - 2)n \geq 10$
- $n \geq 10/(c - 2)$
- Pick  $c = 3$  and  $n_0 = 10$



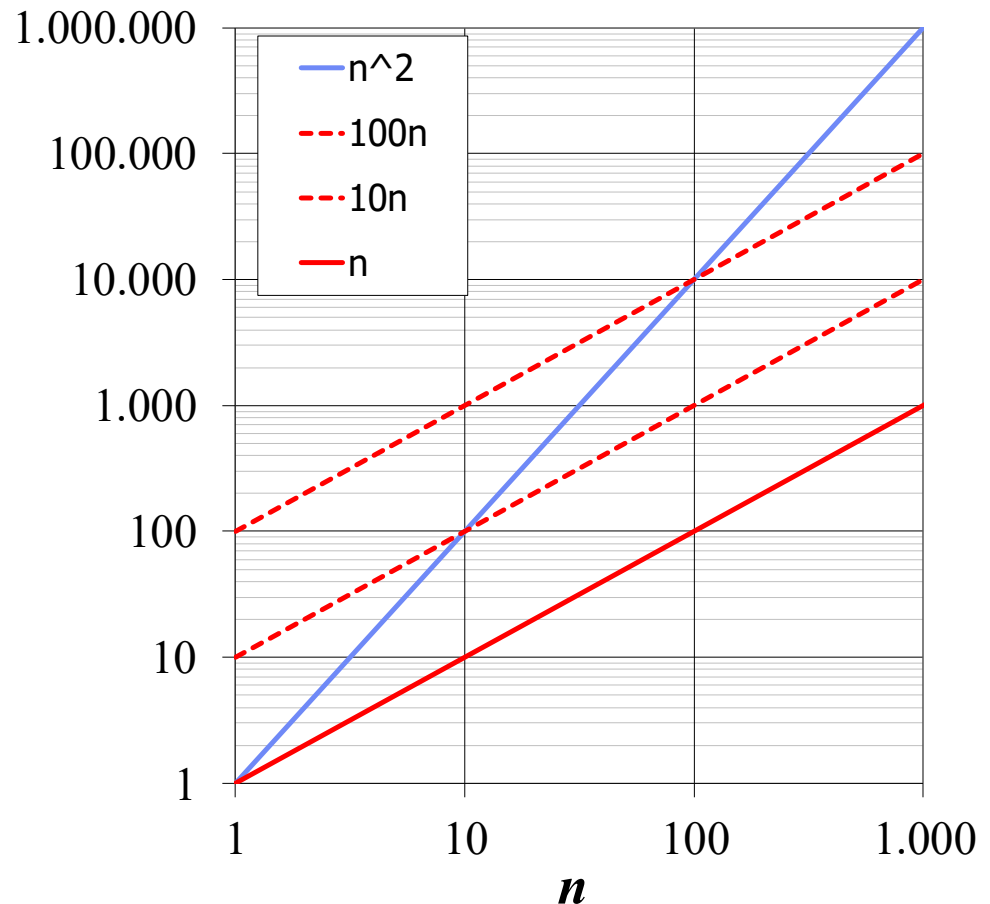
# Big-Oh Example

Example: the function  $n^2$  is not  $O(n)$

–  $n^2 \leq cn$

–  $n \leq c$

The above inequality cannot be satisfied since  $c$  must be a constant



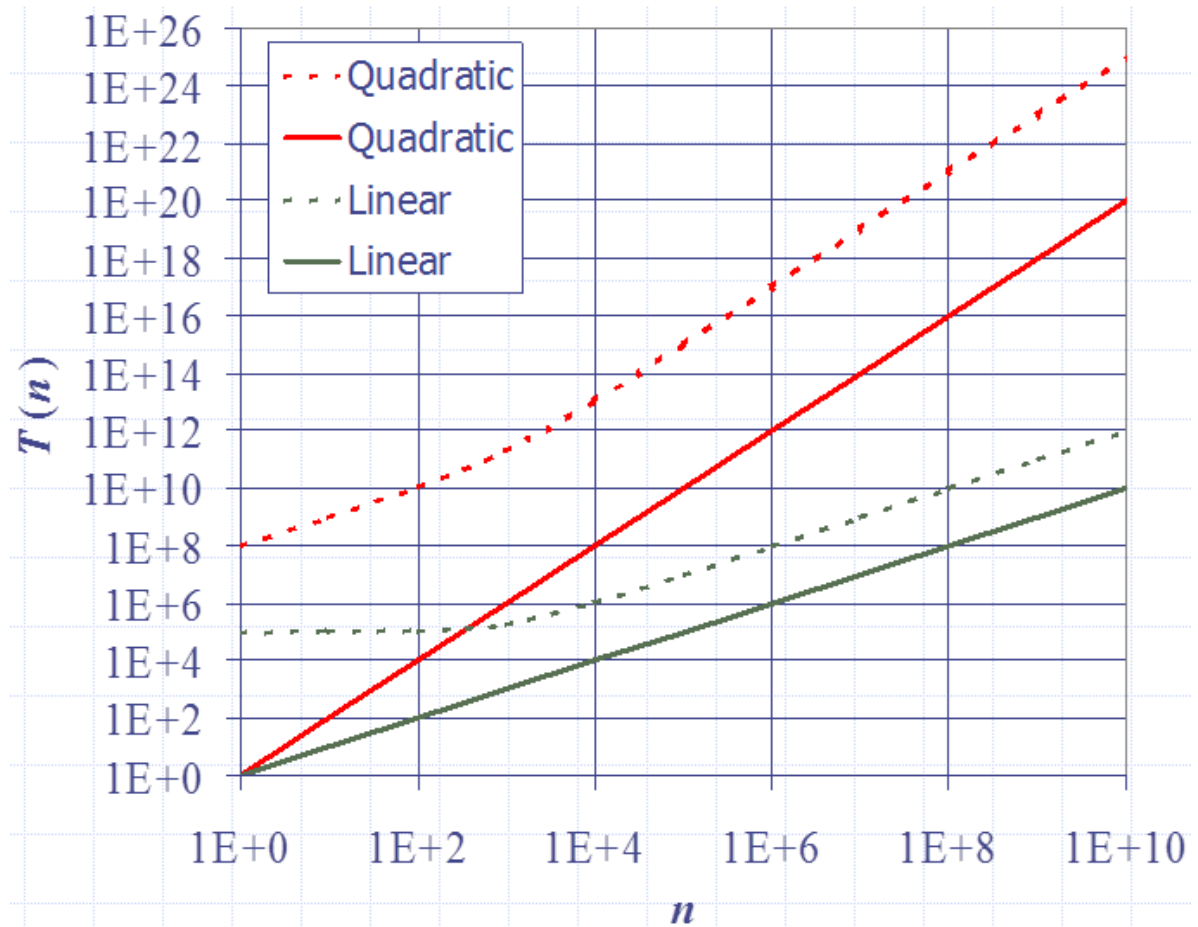
# Constant Factors

The growth rate is not affected by

- constant factors or
- lower-order terms

Examples

- $10^2n + 10^5$
- $10^5n^2 + 10^8n$



# Big-Oh Properties

---

Given the functions

$$F(n) = O(f(n)) \text{ and } G(n) = O(g(n))$$

then

$$F(n) + G(n) = \mathbf{O \left( \max \left( f(n), g(n) \right) \right)}$$

If there are positive constants  $n_1, n_2, c_1, c_2$  such that :

$$n \geq n_1 \rightarrow F(n) \leq c_1 f(n)$$

$$n \geq n_2 \rightarrow G(n) \leq c_2 g(n)$$

and  $n_3 = \max(n_1, n_2)$  ;  $c_3 = \max(c_1, c_2)$

then, for any  $n \geq n_3$ :

$$F(n) + G(n) \leq c_1 f(n) + c_2 g(n)$$

$$F(n) + G(n) \leq c_3 (f(n) + g(n))$$

$$F(n) + G(n) \leq c_3 \max(f(n), g(n))$$

# Big-Oh Properties

---

1.  $O(f) + O(g) = O(f + g) = O(\max(f, g))$   
Ex:  $O(n^2) + O(\log n) = O(n^2)$
2.  $O(f) \times O(g) = O(f \times g)$   
Ex:  $O(n^2) \times O(\log n) = O(n^2 \log n)$
3.  $O(cf) = O(f)$  with  $c$  constant  
Ex:  $O(3n^2) = O(n^2)$
4.  $F = O(f)$   
Ex:  $3n^2 + \log n = O(3n^2 + \log n) = O(n^2)$
5.  $O(n^g) < O(n^{g+k})$

# Determining Big Oh

---

- In practice, it is difficult (if not impossible) to predict accurately the runtime of an algorithm or program
- A method's running time is the sum of time needed to execute sequence of statements, loops, etc. within method
- It is only need to identify one or more **key operations** (frequent operations or time consuming operations) and determine the number of times that they run
- For algorithmic analysis, the largest component dominates **(and constant multipliers are ignored)**

# Determining Big Oh: simple loops

---

For simple loops, determine how many times the loop executes as a function of input size:

- Iterations dependent on a variable  $n$
- Complexity of operations within loop

```
Algorithm arrayMax (A, n)
```

```
{  currentMax ← A[0];
```

```
    for (i ← 1 ; i < n ; i++)
```

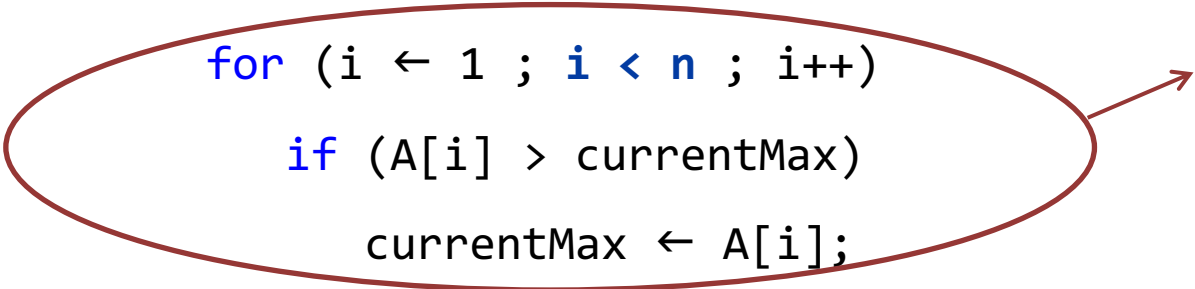
```
        if (A[i] > currentMax)
```

```
            currentMax ← A[i];
```

```
    return currentMax ;
```

```
}
```

$O(n)$





# Primitive Operations

---

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important
- Assumed to take a constant amount of time  $O(1)$

Examples:

- Evaluating an expression
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method

Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

# Determining Big Oh: not so simple loops

---

Not always simple iteration and termination criteria

- Iterations dependent on a function of  $n$

**Example:** Count the number of 1s in a binary representation of a number  $n$

**Algorithm** numberOfOnes ( $n$ )

```
{   count  $\leftarrow$  0 ;
```

```
   while ( $n > 0$ )
```

```
       count  $\leftarrow$  count +  $n \bmod 2$ ;
```

```
        $n \leftarrow n / 2$ ;
```

```
   return count ;
```

```
}
```

$O(?)$

In algorithmic analysis, the log of  $n$  is the number of times you can split  $n$  in half

# Determining Big Oh: nested loops

---

Nested loops (**dependent or independent**) **multiply**

The outer loop have a multiplicative effect on the operations in the internal cycle

```
for (i ← 0 ; i < n ; i++)  
  for (j ← 0 ; j < n ; j++)  
    k ← k+1
```

→ O(?)

```
for (i ← 0 ; i < n ; i++)  
  for (j ← 0 ; j < i ; j++)  
    k ← k+1
```

→ O(?)

# Determining Big Oh: nested Loops

---

```
for (i ← 0 ; i < n ; i++)  
    A[i] ← 0 ;
```

```
for (i ← 0 ; i < n ; i++)  
    for (j ← 0 ; j < n ; j++)  
        k++
```

→  $O(?)$

```
for (h ← n; h > 0; h ← h/2){  
    for (i ← 0 ; i < n ; i++)  
        k++ ;  
}
```

→  $O(?)$

# Determining Big Oh: Calling Functions

---

When one function calls another, big-Oh of calling function also considers big-Oh of called function and how many times embedded function(s) are called

```
private static void removeElem (ArrayList<Integer> v, int elem)
{
    int i = v.size()-1;    O(?)

    while (i >= 0){
        if (elem == v.get(i).intValue() ) O(?)
            v.remove(i); O(?)
        i = i - 1 ;
    }
}
```

removeElem:  $O(?)$

---

# **Non-deterministic Algorithms**

# Deterministic vs. non-deterministic algorithms

---

- An algorithm is *deterministic* if at each step there is only one choice for the next step given the values of the variables at that step
  - mean of the elements of a vector
  - maximum element of a vector
  - multiplication of two matrices
- An algorithm is *non-deterministic* if there are steps that may be not executed, depending on the values in data structures
  - search a value in a vector
  - ordering the values of a vector

# Non-deterministic algorithms

---

To correctly analyse the complexity of a non deterministic algorithm it is necessary to consider:

- **Worst case analysis:**

We calculate **upper bound on running time** of an algorithm. We must know the case that causes maximum number of operations to be executed

- **Average case analysis:**

We take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs

- **Best case analysis:**

We calculate **lower bound on running time** of an algorithm. We must know the case that causes minimum number of operations to be executed



# Sequential search (non-deterministic algorithm)

---

**Example:** Search for a target  $x$  in an unordered array  $A$  of  $n$  elements

```
Algorithm int sequentialSearch (T v[], int n, T x) {  
    i ← 0 ;  
    while (i < n) {  
        if (v[i] == x)  
            return i ;  
        i ← i + 1 ;  
    }  
    return -1 ;  
}
```

# Sequential search (non-deterministic algorithm)

---

Sequential search on an unsorted array of length  $n$ , what is:

- **Best case?**
- **Worst case?**
- **Average case?**

# Sequential search (non-deterministic algorithm)

---

**Best case:** occurs when the element to be searched is present at the first location

$$T(n) = O(1)$$

**Average case:** We must know (or predict) the distribution of cases. Assuming all cases are **uniformly distributed** (including the case of x not being present in array)

$$T(n) = \frac{\sum_{i=1}^{n+1} O(i)}{n+1} = O\left(\frac{(n+1) \times (n+2)}{2 \times (n+1)}\right) \Rightarrow T(n) = O(n)$$

**Worst case:** occurs when the element to be searched is not present in the array

$$T(n) = O(n)$$

# What to analyse?

---

- The average case analysis is often difficult to determine in most of the practical cases and it is rarely done
- In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs
- Most of the times, the average case analysis is equal to the worst case analysis
- We focus on the best and on the worst case analysis
  - Easier to analyse
  - Guarantees a lower and an upper bound on the running time of an algorithm which is a good information

# Space complexity - Sequential search

---

```
Algorithm int sequentialSearch (T v[], int n, T x) {  
    i ← 0 ;  
    ...  
    return -1 ;  
}
```

## Total space:

return address	2 bytes
v address	2 bytes
n address	2 bytes
x address	2 bytes
local variable i	<u>2 bytes</u>
	10 bytes

$$S(n) = O(1)$$

---

# **Recursive Algorithms**

# Big Oh: Recursion

---

For recursion it is necessary to determine:

- how many times the function will be executed?
- which is the complexity of the body function
- Multiply these together

```
int recursivefunction (...) {
```

$O(?)$

```
    recursivefunction (...);
```

```
}
```

# Factorial function

---

```
private static double factiter (double num){  
    double res=1 ;  
    for (int i = 1; i <= num; i++)  
        res = res * i ;  
    return res ;  
}
```

$T(n) = O(?)$

$S(n) =$

```
private static double factrecurs (double num){  
    if (num == 1)  
        return 1 ;  
    else  
        return num * factrecurs(num-1) ;  
}
```

$T(n) = O(?)$

$S(n) =$



# Fibonacci Iterative

---

```
Algorithm int fibiter (int n)
{
    int fib = 0;        // current
    int ant = 0;        // previous

    for (int i = 1; i <= n; i++) {

        if (i == 1) {
            fib = 1;
            ant = 0;
        } else {
            fib += ant;
            ant = fib - ant;
        }
    }
    return fib;
}
```

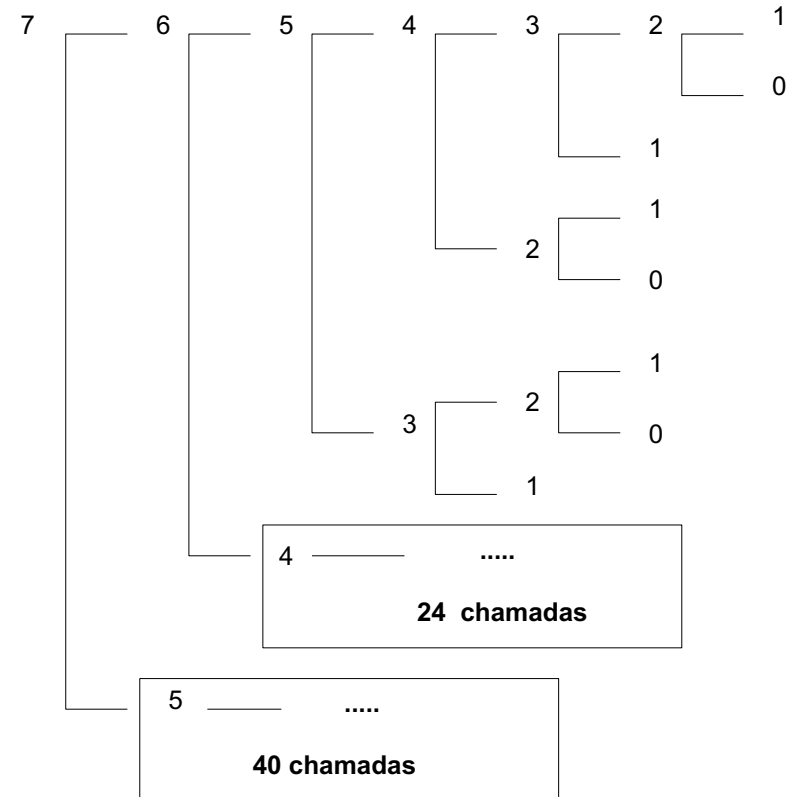
$T(n) = O(?)$

# Fibonacci recursive

```
Algorithm int fibrecurs (int n) {
    if (n ≤ 1)
        return n ;
    else
        return fib(n-1) + fib(n-2);
}
```

n	k	Function
2	3	$\leq 2^{n-1} + 1$
3	5	$\leq 2^{n-1} + 1$
4	9	$\leq 2^{n-1} + 1$
5	15	$\leq 2^{n-1} + 1$
6	24	$\leq 2^{n-1} + 1$
....	...	$\leq O(2^n)$

$T(n) = O(2^n)$       Exponential



# Torres de Hanói

```
Algorithm void Hanoi (int N, Stack<T> TA, Stack<T> TB, Stack<T> TC) {  
    if (N = 1)  
        Move disk TA → TC  
    else  
        Hanoi (N-1, TA, TC, TB)  
        Move disk TA → TC  
        Hanoi (N-1, TB, TA, TC)  
}
```

N (N. of disks)	K (N. movements)
1	1
2	3
4	15
8	63
16	...
...	...
	$2^n - 1$

$$T(n) = O(2^n)$$

Exponencial

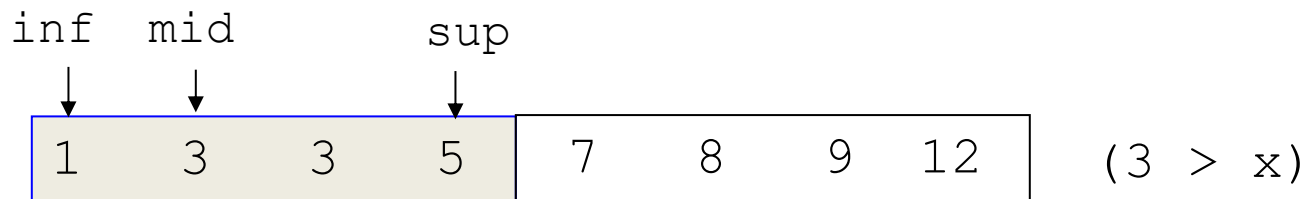
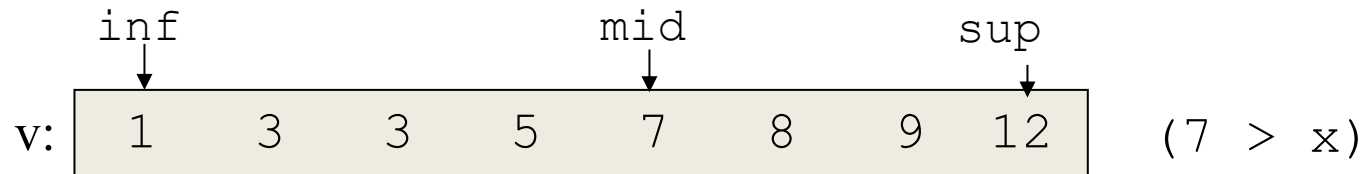
---

# Searching and Sorting Algorithms

# Binary search

**Example:** Search for a target  $x$  in an **ordered array**  $A$  of  $n$  elements

**Worst case:** Element to be searched is not present ,  $x = 2$



# Binary search

---

```
Algorithm int binarySearch (T v[], int n, T x) {  
    inf ← 0 ;  
    sup ← n ;  
    while (inf <= sup) {  
        mid ← (inf+sup)/2  
        if (v[mid] == x)  
            return mid ;  
        else  
            if (v[mid] > x)  
                sup ← mid-1 ;  
            else  
                inf ← mid+1 ;  
    }  
    return -1 ;  
}
```

# Binary search

---

- Through each iteration of Binary Search, the size of the admissible range is halved
- For an array of size  $N$ , it eliminates  $\frac{1}{2}$  until 1 element remains  
 $N, N/2, N/4, N/8, \dots, 4, 2, 1$

- How many divisions does it take?
- Or, putting it more simple, how many times do I have to multiply by 2 to reach  $N$ ?

$1, 2, 4, 8, \dots, N/4, N/2, N$

$$2^x = N \Rightarrow x = \log_2 N$$

Worst case:  **$T(n) = O(\log n)$**

# Sorting

---

Basic problem: order elements in a vector

Need to know relationship between data elements: availability of **comparator** for elements

- Simple Sort Algorithms:
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
- Complex Sort Algorithms
  - Count Sort
  - Shaker Sort
  - Shell Sort
  - Heap Sort
  - Merge Sort
  - Quick Sort



# Selection Sort

Sorted

Unsorted

23	78	45	8	32	56
----	----	----	---	----	----

Original Vector

8	78	45	23	32	56
---	----	----	----	----	----

After pass 1

8	23	45	78	32	56
---	----	----	----	----	----

After pass 2

8	23	32	78	45	56
---	----	----	----	----	----

After pass 3

8	23	32	45	78	56
---	----	----	----	----	----

After pass 4

8	23	32	45	56	78
---	----	----	----	----	----

After pass 5

# Selection Sort

---

```
Algorithm void selectionSort (T v[], int n) {  
    for (int i ← 0; i < n-1; i++) {  
        min ← i  
        for (int j ← i+1; j < n; j++){  
            if (v[j] < v[min])  
                min = j  
        swap(v[i], v[min])  
    }  
}
```

The outer for loop executes  $n-1$  times

The inner for loop executes the size of the unsorted part minus 1,  
from 1 to  $n-1$

Deterministic Algorithm:  $O(?)$

# Bubble Sort

---

23	78	45	8	32	56
----	----	----	---	----	----

Original Vector

23	78	8	45	32	56
----	----	---	----	----	----

23	8	78	45	32	56
----	---	----	----	----	----

8	23	78	45	32	56
---	----	----	----	----	----

...

8	23	32	78	45	56
8	23	32	45	78	56
8	23	32	45	56	78

# Bubble Sort

---

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

Deterministic algorithm:  $O(?)$

# Bubble Sort Optimized

---

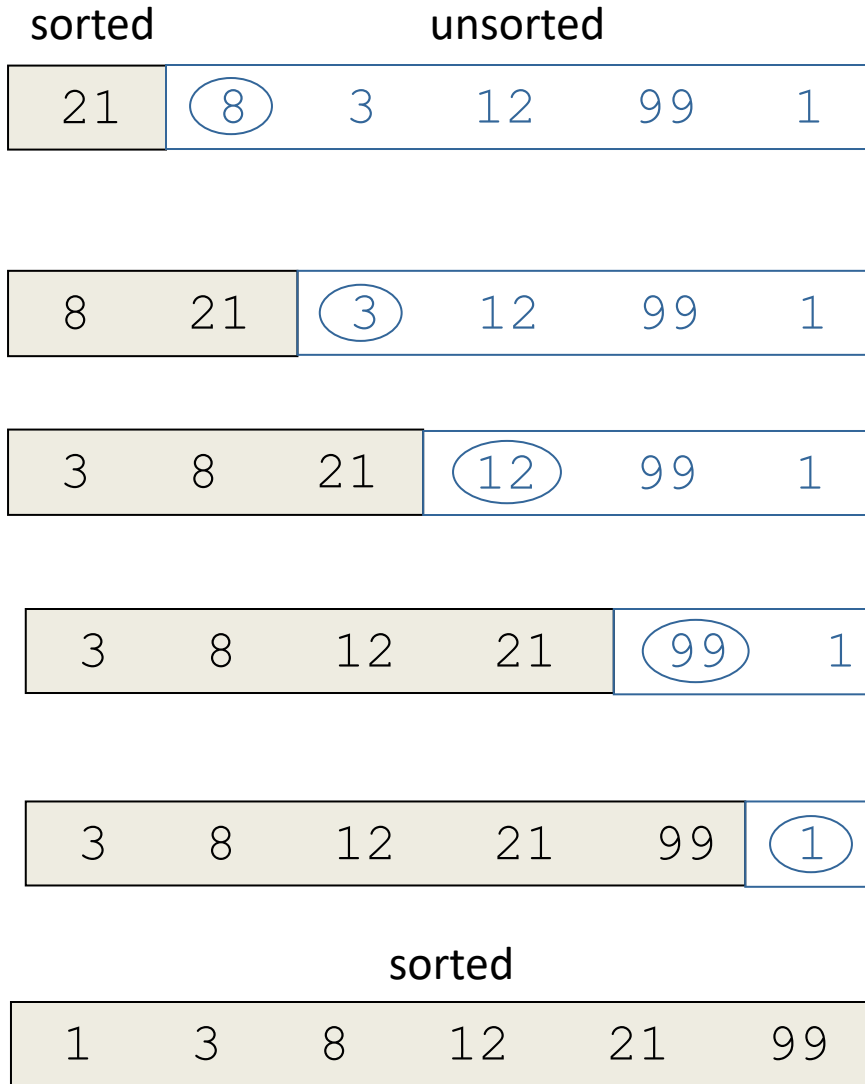
```
Algorithm void bubbleSort (T v[], int n) {  
    swap ← true  
    for (i ← 0; (i < n-1 && swap); i++) {  
        swap ← false  
        for (j ← n-1; j > i; j--)  
            if (v[j-1] > v[j]) {  
                swap(v[i], v[j-1])  
                swap ← true  
            }  
        }  
    }  
}
```

Best case:  $O(?)$

Worst case:  $O(?)$

# Insertion Sort

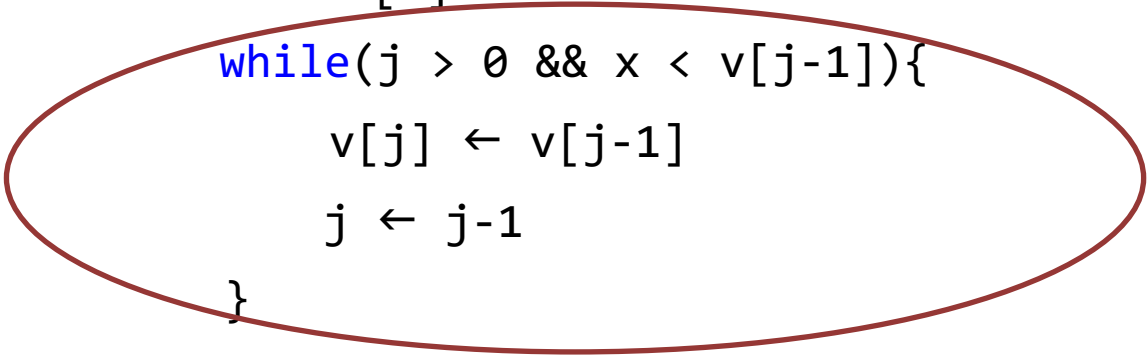
---



# Insertion Sort

---

```
Algorithm void insertionSort (T v[], int n) {  
    for (i ← 1 ; i < n ; i++) {  
        j ← i  
        x ← v[i]  
        while(j > 0 && x < v[j-1]){  
            v[j] ← v[j-1]  
            j ← j-1  
        }  
        v[j] ← x ;  
    }  
}
```



**Sub problem:** Insertion of an element in a sorted vector

Best case:  $O(?)$

Worst case:  $O(?)$

# Divide-and-conquer Pattern

---

Consists of the following three steps:

1. **Divide:** If the input size is greater than a certain threshold (say, one or two elements) divide the input data into two or more disjoint subsets
2. **Conquer:** Recursively solve the **sub problems** associated with the subsets
3. **Combine:** Take the solutions to the sub problems and merge them into a solution to the original problem



# Merge-Sort Algorithm

---

Merge-Sort is a recursive algorithm that uses the **algorithmic design pattern divide-and-conquer**

To sort a sequence  $S$  with  $n$  elements the Merge-Sort algorithm proceeds as follows:

- 1. Divide:** If  $S$  has zero or one element, return  $S$  immediately  
Otherwise, divide  $S$  into two sequences,  $S_1$  and  $S_2$ , each containing about half of the elements of  $S$
- 2. Conquer:** Recursively sort sequences  $S_1$  and  $S_2$
- 3. Combine:** Put the elements back into  $S$  by merging the sorted sequences  $S_1$  and  $S_2$  into a sorted sequence

# Merge-Sort Algorithm

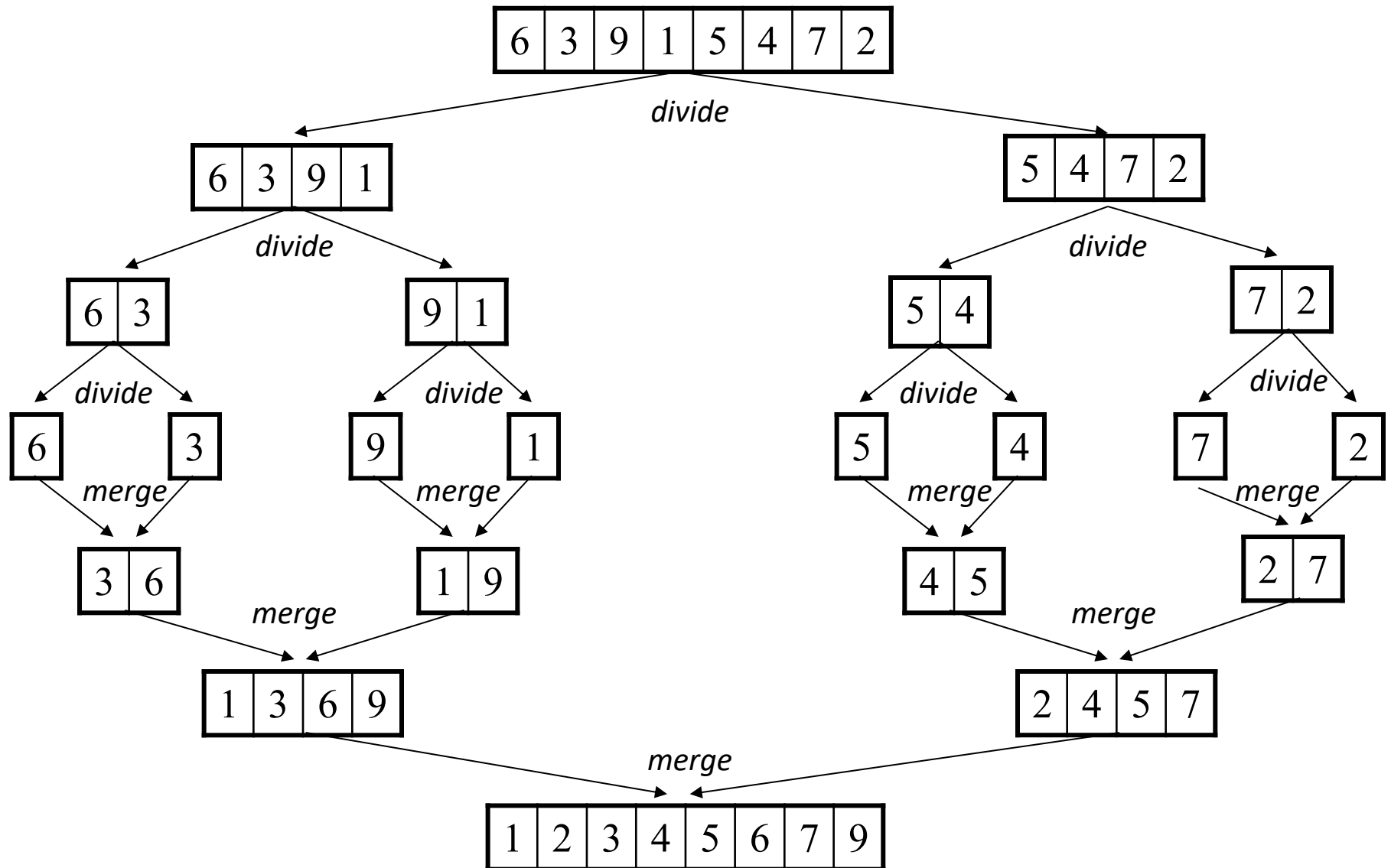
---

**Input:** Sequence S with n elements

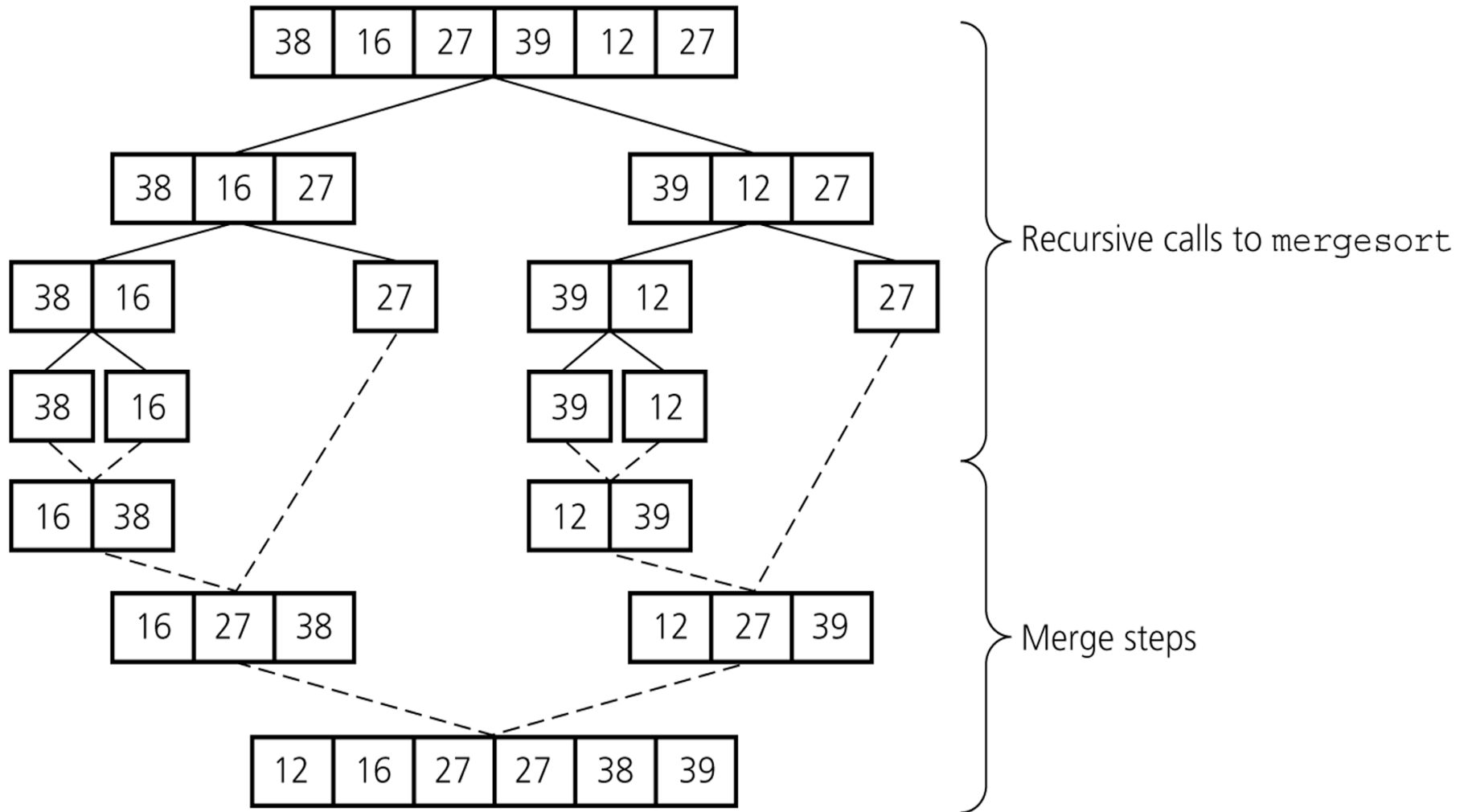
**Output:** Sequence S sorted

```
Algorithm void mergeSort (T S[], int n) {  
  
    if (n >= 2) {  
        int mid = n/2;                // index of midpoint  
        T S1[] = S[0,..,mid]  
        T S2[] = S[mid,..,n]  
        mergeSort(S1,n1);  
        mergeSort(S2,n2);  
        merge(S1, S2, S)    //merge two sorted sequences  
    }  
}
```

# Merge-Sort example

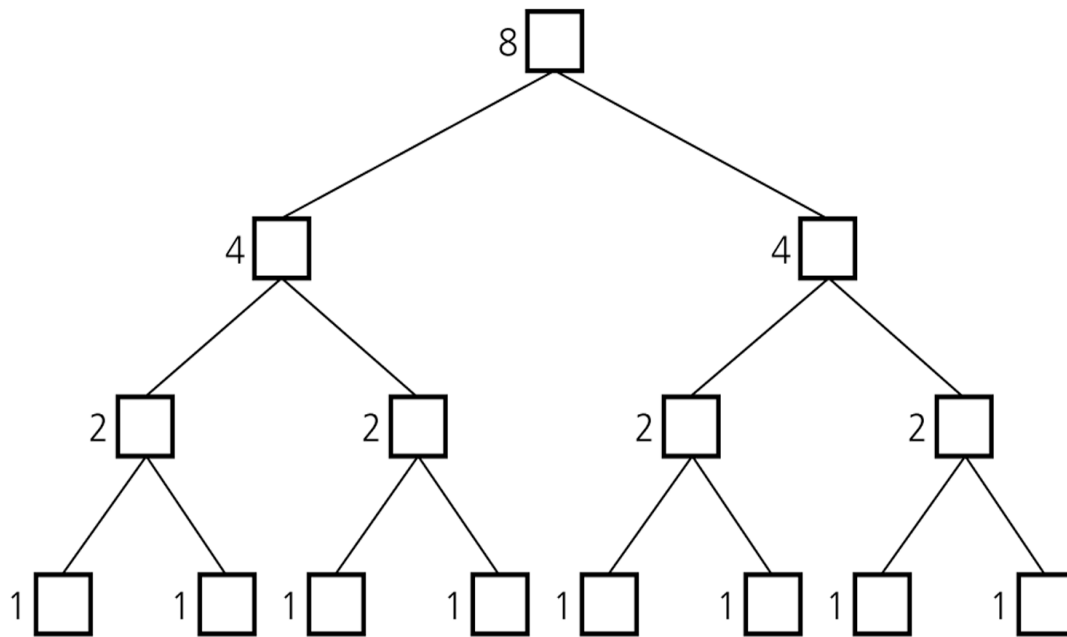


# Merge-Sort another example



# Analysis of Merge-Sort

Levels of recursive calls to Merge-Sort, given an array of eight items



Level 0: mergesort 8 items

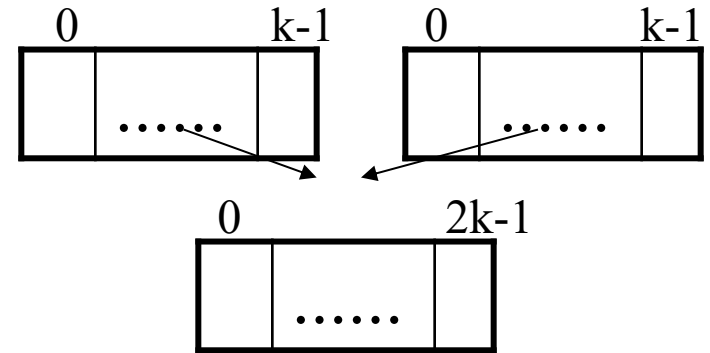
Level 1: 2 calls to mergesort with 4 items each

Level 2: 4 calls to mergesort with 2 items each

Level 3: 8 calls to mergesort with 1 item each

# Analysis of Merge-Sort

Merging two sorted arrays of size  $k$



- **Best-case:**
  - All the elements in the first array are smaller (or larger) than all the elements in the second array.
  - The number of moves:  $2k + 2k$
  - The number of key comparisons:  $k$
- **Worst-case:**
  - The number of moves:  $2k + 2k$
  - The number of key comparisons:  $2k-1$

# Analysis of Merge-Sort

---

- Merge-Sort is an extremely efficient algorithm with respect to time
  - Both, worst and average cases are  $O(n \log n)$
- Merge sort requires additional memory with size  $N$
- Usually Merge-Sort is not used for main memory sorting, only for external memory sorting

# QuickSort Algorithm

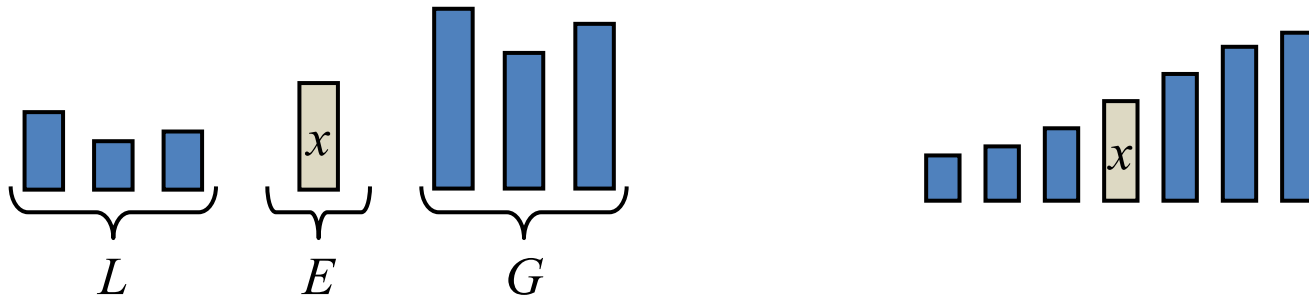
---

Like Merge-Sort, Quicksort is also based on the ***divide-and-conquer*** paradigm

But it uses this technique in a somewhat opposite manner, as all the hard work is done *before* the recursive calls

It works as follows:

1. First, it partitions an array into two parts
2. Then, it sorts the parts independently
3. Finally, it combines the sorted subsequences by a simple concatenation





# QuickSort Algorithm – Basic Idea

---

- Pick one element in the array, which will be the **pivot**
- Make one pass through the array, called a partition step, re-arranging the entries so that:
  - entries smaller than the pivot are to the left of the pivot
  - entries larger than the pivot are to the right
- Recursively apply quicksort to the left and right parts of the array
- **No merge step**, at the end all the elements are in the proper order



# QuickSort Algorithm

---

```
Algorithm void quickSort (T S[], int left, int right) {
    pivot ← S[(left+right)/2]
    i ← left
    j ← right
    while (i <= j){
        while (S[i] < pivot)
            i++

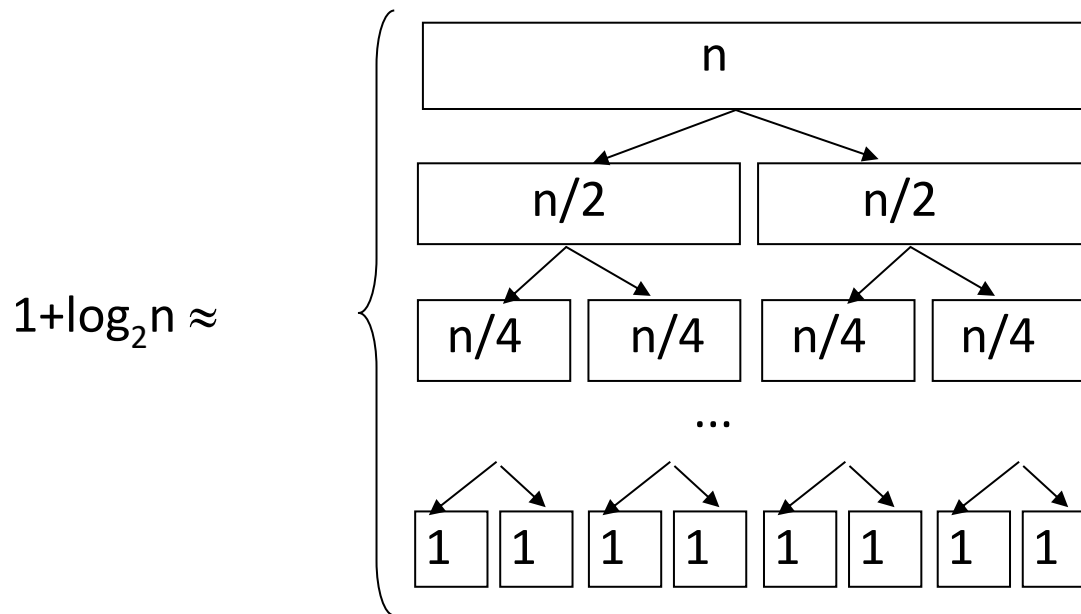
        while (S[j] > pivot)
            j--

        if (i <= j) {
            swap(S[i],S[j])
            i++
            j--
        }
    }
    if (left < j)
        quickSort(S,left,j)
    if (right > i)
        quickSort(S,i,right)
}
```

# Analysis of QuickSort

## Best and Average-case:

Happens when the pivot is the median of the array, the left and the right parts have same size

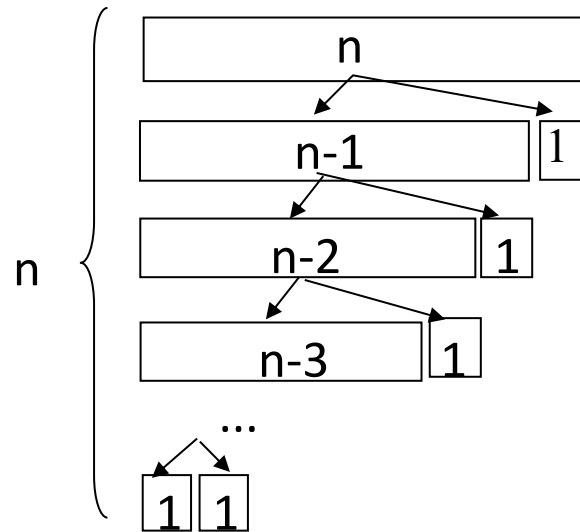


- There are  **$\log N$**  partitions, and to obtain each partitions we do  **$N$**  comparisons (and not more than  **$N/2$**  swaps)
- Hence the complexity is  **$O(n \log n)$**

# Analysis of QuickSort

## Worst Case:

- Happens when the **pivot** is the **smallest (or the largest)** element
- Then one of the partitions is empty, and we repeat recursively the procedure for  $N-1$  elements



The number of key comparisons =  $n-1 + n-2 + \dots + 1$

$$= \frac{n^2}{2} - \frac{n}{2}$$

→  $O(n^2)$

The number of swaps =  $n-1 + n-1 + n-2 + \dots + 1$

→  $O(n^2)$

So, Quicksort is  **$O(n^2)$**  in worst case

# Analysis of QuickSort

---

- One of the fastest algorithms on average  $O(n \log n)$
- Does not need additional memory (the sorting takes place in the array - this is called **in-place processing** )
- The above advantages compensate for the rare occasions when it runs with  $O(n^2)$

# Faster computer vs. better Algorithm

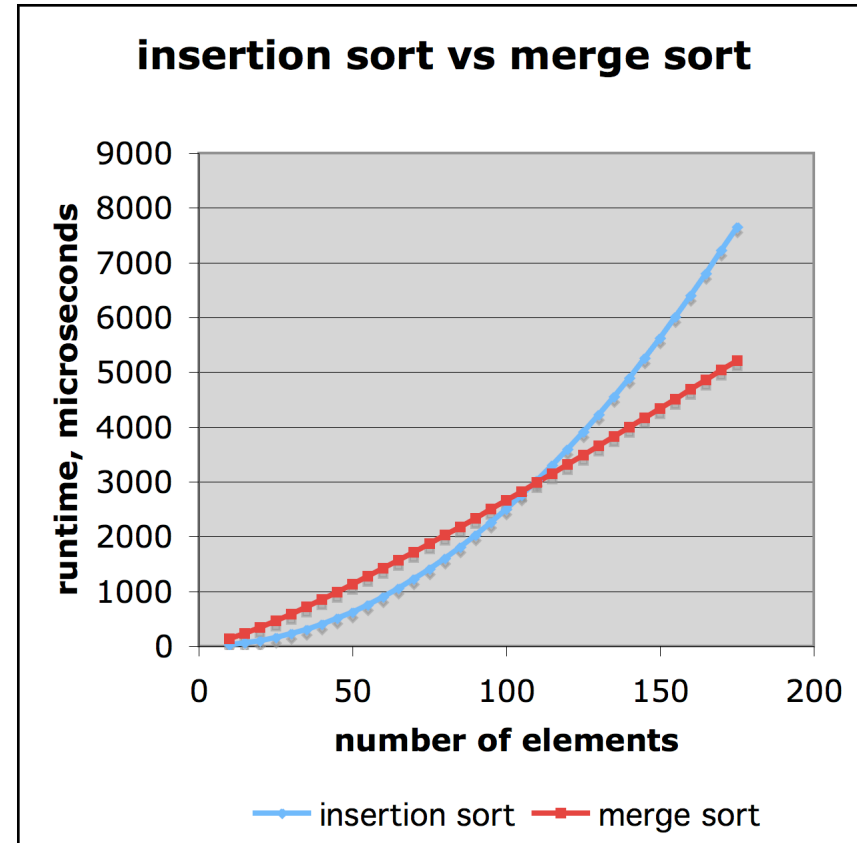
insertion sort is  $O(n^2)$

merge sort is  $O(n \log n)$

Sort a million items:

insertion sort takes roughly 70 hours  
while

merge sort takes roughly 40 seconds



**Algorithmic improvement is always more clever than hardware improvement**

---

# **Complexity Analysis**

## **JAVA Collections Framework**



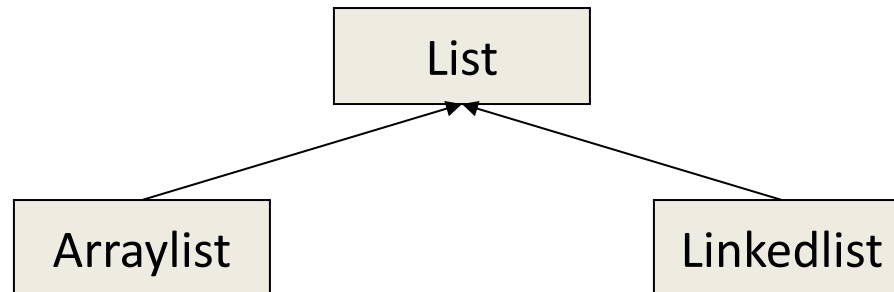
# List<E> Interface

---

In addition to the core services inherited from the root collection interface, the list interface offers

- Positional access
- Search
- Customized Iteration
- Range-view

## List Implementations



# ArrayList and LinkedList

---

- **ArrayList:**
  - Constant time positional access
  - One tuning parameter, the initial capacity
- **LinkedList:**
  - Stores each element in a node
  - Each node stores a link to the next and previous nodes
  - Insertion and removal are inexpensive
    - just update the links in the surrounding nodes
  - Linear traversal is inexpensive
  - Random access is expensive
    - Start from beginning or end and traverse each node while counting

# List<E> Implementations

---

Method	Complexity	
	ArrayList<E>	LinkedList<E>
size()	O(1)	O(1)
isEmpty()	O(1)	O(1)
get(i)	O(1)	O(n)
set(i, e)	O(1)	O(n)
add(e)	O(1)	O(1)
add(i,e)	O(n)	O(n)
addFirst(e)		O(1)
addLast(e)		O(1)
remove(i)	O(n)	O(n)
remove(e)	O(n)	O(n)

# ArrayList vs. LinkedList

---

A list can grow or shrink dynamically. An array is fixed once it is created

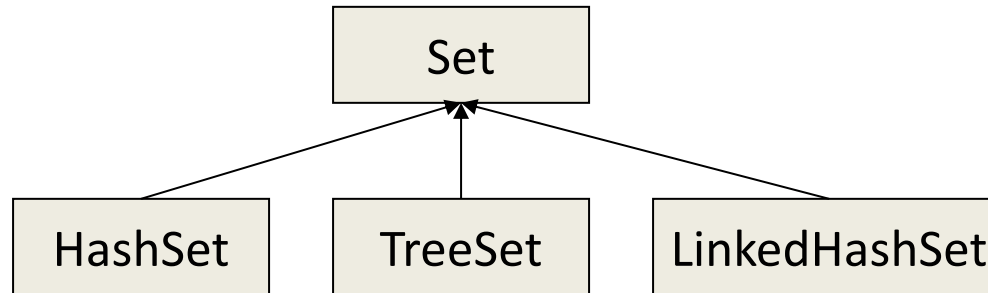
- ArrayList most efficient if:
  - need to support **random access through an index**
  - without inserting or removing elements from any place other than the end
- LinkedList most efficient if:
  - application requires **insertion or deletion of elements** from **any place** in the list

# Set<E> Interface

---

- The Set interface extends Collection and contains no methods other than those inherited from Collection
- It adds the functionality of restricting duplicate elements

## Set Implementations



	HashSet	TreeSet	Linked HashSet
<b>Storage Type</b>	Hash Table	Red-Black Tree	Hash Table with a Linked List
<b>Order of Iteration</b>	No guarantee of order of iteration	Order based	Orders elements based on insertion

# Set<E> Implementations

---

Method	Complexity		
	HashSet<E>	TreeSet<E>	LinkedHashSet<E>
size()	O(1)	O(1)	O(1)
isEmpty()	O(1)	O(1)	O(1)
clear()	O(n)	O(n)	O(n)
add(e)	O(1)	O(logn)	O(1)
remove(e)	O(1)	O(logn)	O(n)
contains(e)	O(n)	O(logn)	O(n)
addAll()	O(n)	O(nlogn)	O(n)
retainAll()	O(n)	O(nlogn)	O(n)
removeAll()	O(n)	O(nlogn)	O(n)
containsAll()	O(n <sup>2</sup> )	O(nlogn)	O(n <sup>2</sup> )

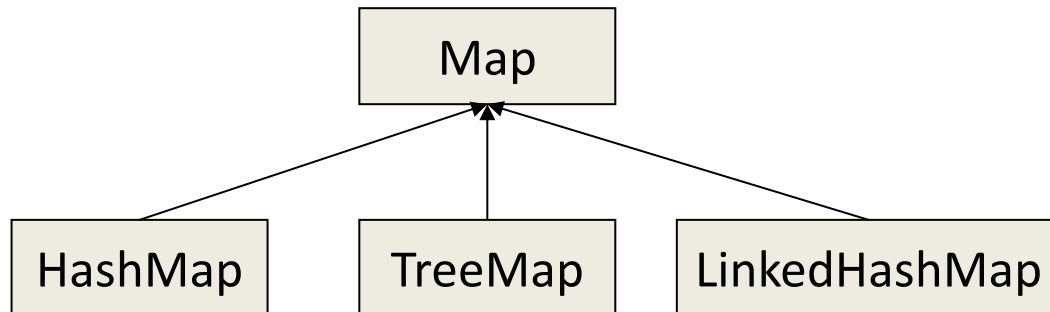
# Map<K,V> Interface

---

Replaces java.util.Dictionary interface

- Keeps association between key and value objects
- Every key in a map has a unique value
- A value may be associated with several keys

## Map Implementations



# Map<K,V> Implementations

Method	Complexity		
	HashMap	TreeMap	LinkedHashMap
size()	O(1)	O(1)	O(1)
isEmpty()	O(1)	O(1)	O(1)
clear()	O(n)	O(n)	O(n)
put(K,O)	O(1)	O(logn)	O(1)
get(K)	O(1)	O(logn)	O(n)
remove(k)	O(1)	O(logn)	O(n)
containsKey (K)	O(n)	O(logn)	O(n <sup>2</sup> )
containsValue (O)	O(n)	O(logn)	O(n <sup>2</sup> )
putAll(M)	O(n)	O(nlogn)	O(n)
keySet()	O(n)	O(n)	O(n)
values()	O(n)	O(n)	O(n)



# How To Select a Container

---

Determine how you access elements

- access by key (any)
- access by integer index

**(any) Map**

**ArrayList**

Determine whether iteration order matters

- elements must be sorted
- elements must stay in order as inserted

**TreeMap**

**LinkedHashMap**

Determine which operations need to be fast

- adding and removing elements must be fast
- Finding elements must be fast (any)

**LinkedList**

**HashMap**

.