Estruturas de Informação

Algorithm Analysis

Fátima Rodrigues
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

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

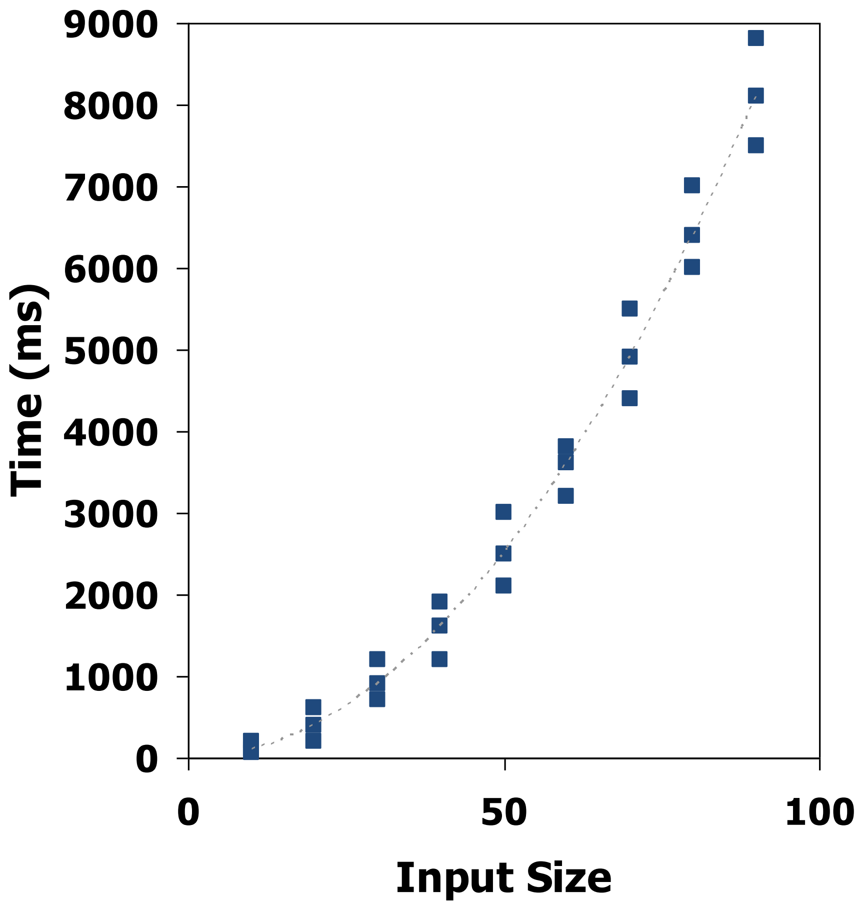
Algorithm Analysis

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

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 inputs size 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:

$$\frac{n^2}{(2n)^2} = \frac{t_1}{t_2}$$

then solve for t_2 …

Algorithm Analysis

Algorithm Analysis only considers the magnitude of time in function of the size of the input data

- The point is to relate the variation of the algorithm execution time with the input size variation
- to predict the growth of the resources required by the algorithm as the size of input data increases

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 ≈ ***1***
- Logarithmic ≈ log ***n***
- Linear ≈ ***n***
- N-Log-N ≈ ***n*** log ***n***
- Quadratic ≈ n^2
- Cubic ≈ n^3

Exponential:

- 2^n
- 3^n
- n!

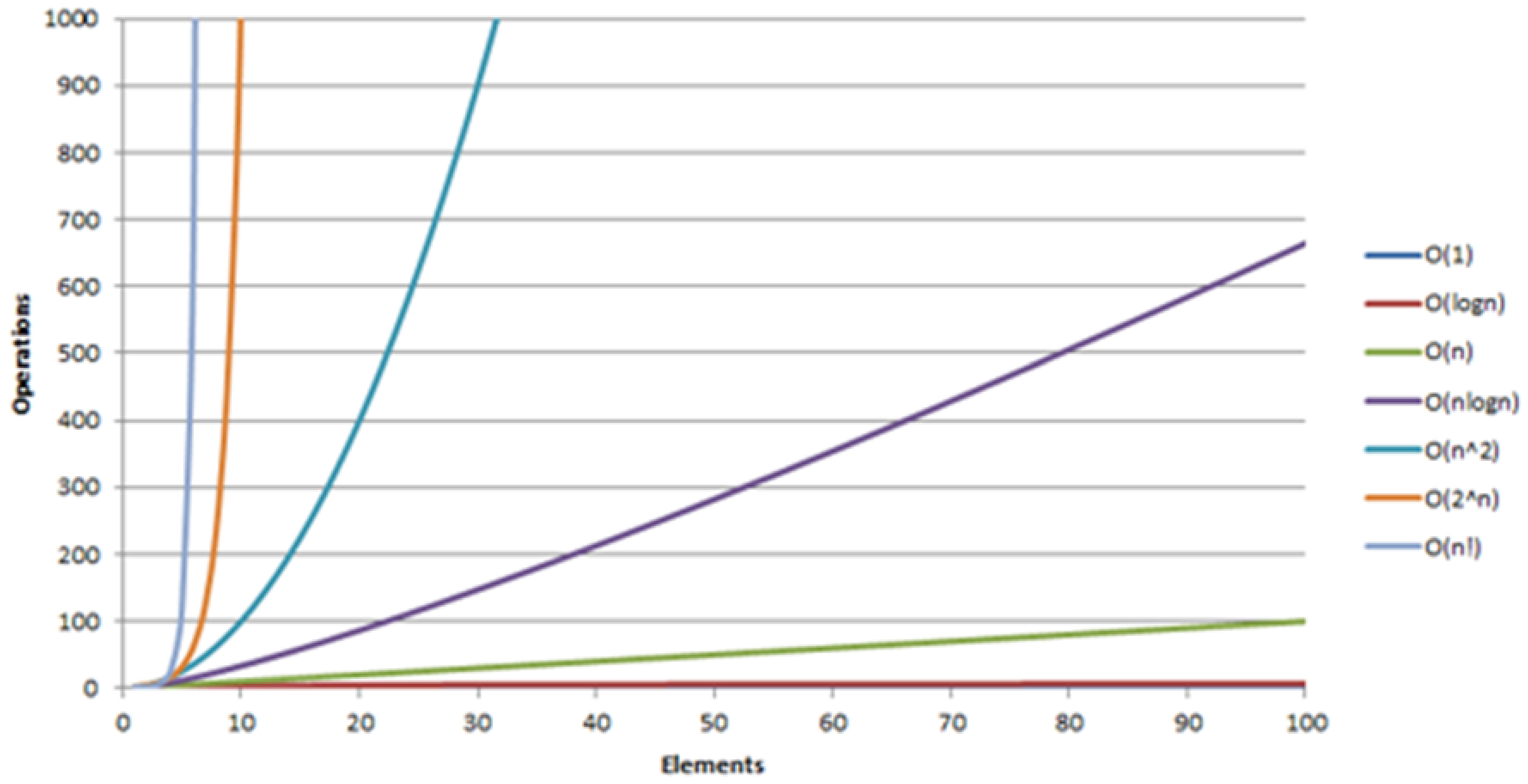
Most common Complexity Orders

Input data: n = 10^5 elements

Execution time of each step k = 10^{-5} seconds = 10 μs

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	3.2×10^{-4} sec
$\log n$	Logarithmic	170 μs
1	Constant	10 μs

Computation Complexities



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	`Algorithm int arrayMax (int A[], int n)`	
2	`{   currentMax ← A[0];`	
3	`    for (i ← 1 ; i < n ; i++)`	
4	`        if (A[i] > currentMax)`	
5	`            currentMax ← A[i];`	
6	`   return currentMax ;`	
	`}`	

Three different Notations O, Ω e Θ

big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

Asymptotic Algorithm Analysis

The asymptotic analysis of an algorithm determines the running time of the algorithm when it runs on large inputs

To perform the asymptotic analysis

- We find the worst-case number of primitive operations executed as a function of the input size
- We express this function with **big-Oh notation**

Example:

- The algorithm arrayMax executes at most **7n-2** primitive operations
- We say that algorithm arrayMax "*is asymptotically at most n*"

The *asymptotic notation or Big-Oh notation* for describing the algorithmic complexity is $f(n) = O(n)$

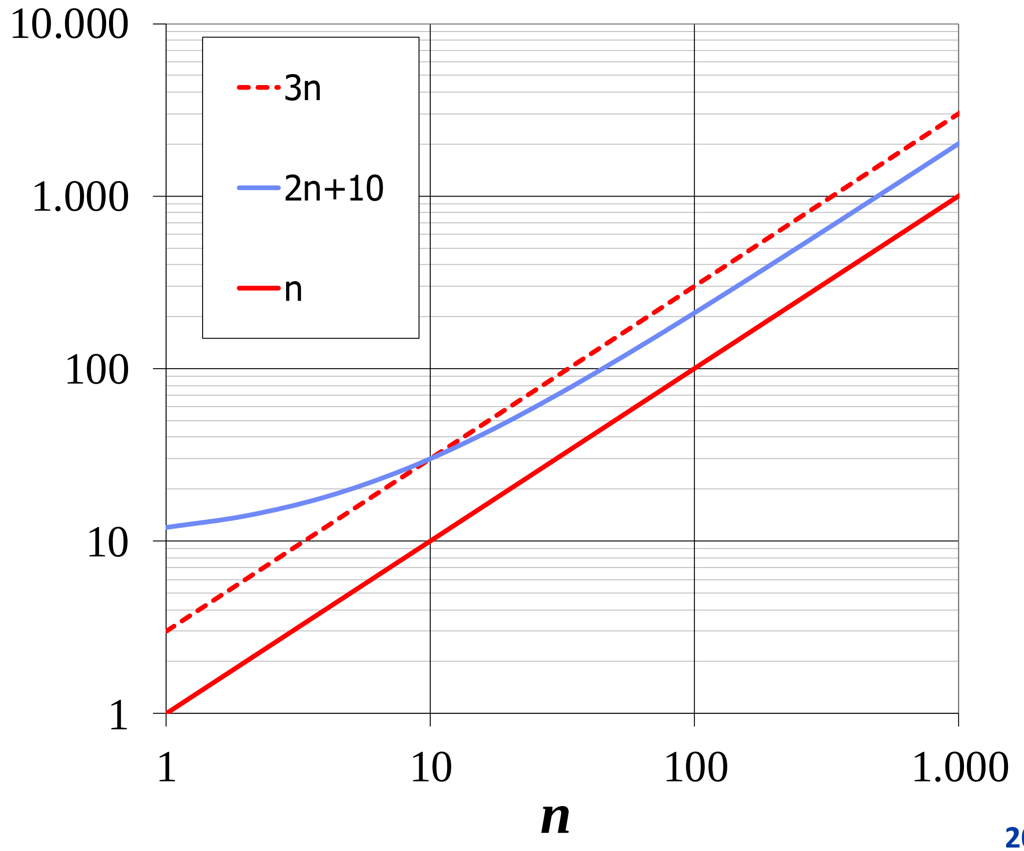
Big-Oh Notation

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 $n_0 = 10$

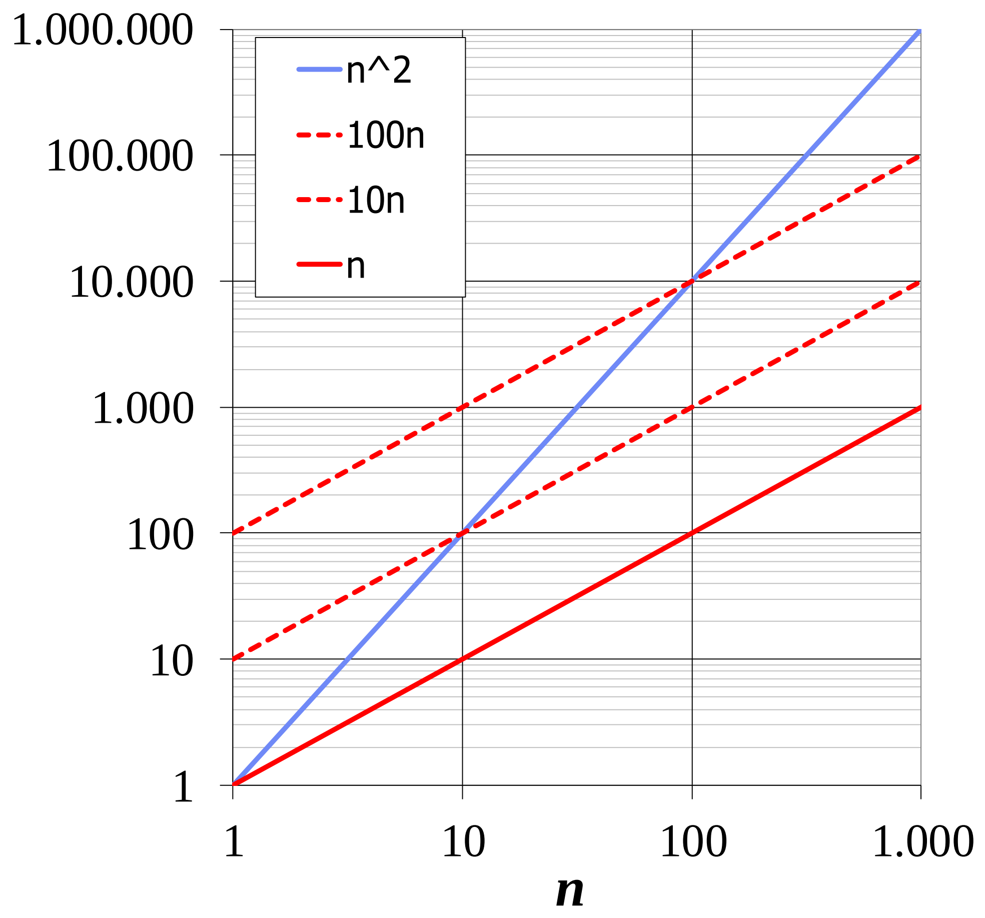


Big-Oh Example

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



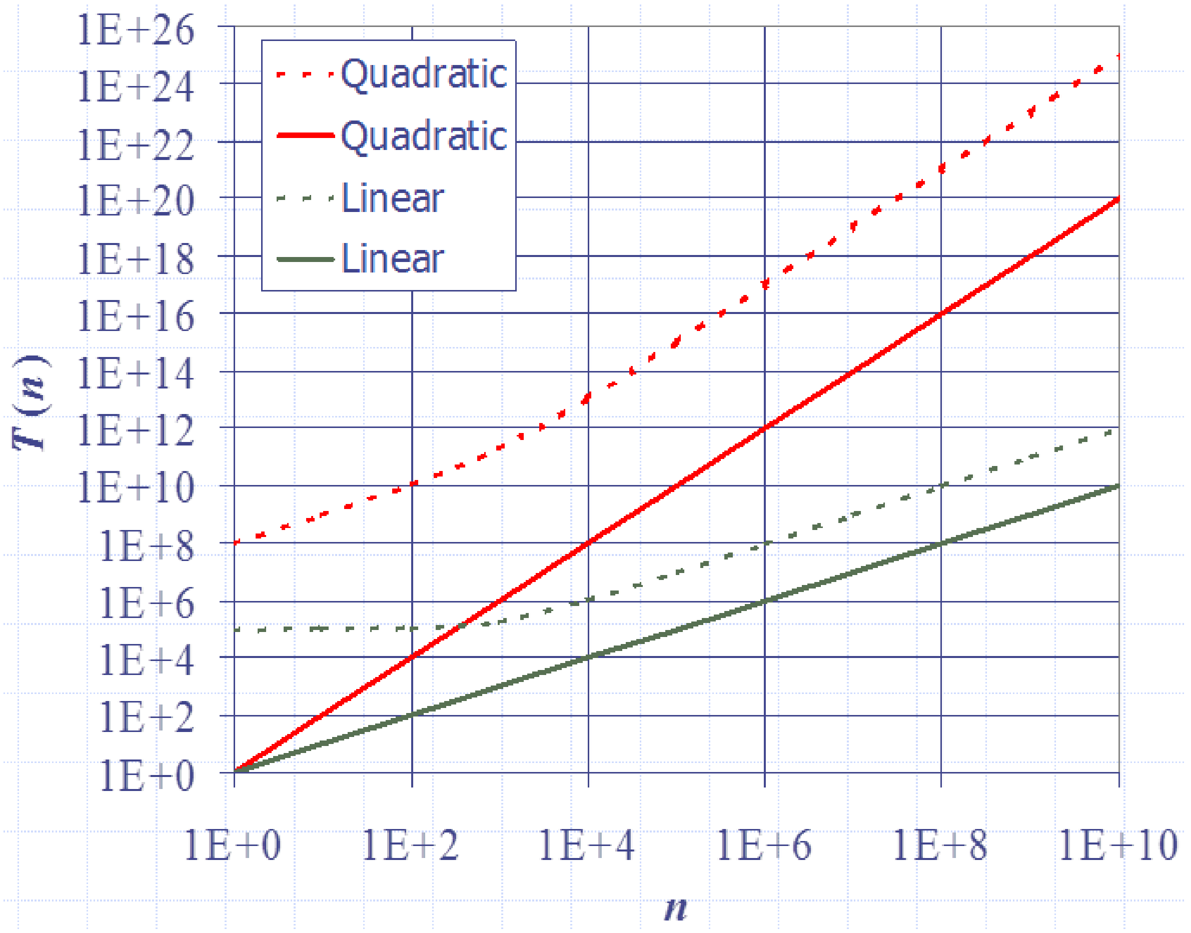
Constant Factors

The growth rate is not affected by

- constant factors or
- lower-order terms

Examples

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



Big-Oh Properties

Given the functions

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

then

$$F(n) + G(n) = O(\max(f(n), g(n)))$$

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. $O(n^g) < O(n^{g+k})$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "***f***(***n***) is ***O***(***g***(***n***))" means that the growth rate of ***f***(***n***) is no more than the growth rate of ***g***(***n***)
- We can use the big-Oh notation to rank functions according to their growth rate

Example: analysis of a given method shows its execution time as

$$\underbrace{\underline{8n}}_{\text{Single loop}} + \underbrace{\underline{3n^2}}_{\text{Nested loop}} + \underbrace{\underline{23}}_{\substack{\text{Constant time}\\ \text{Statements}}}$$

Don't write $O(8n + 3n^2 + 23)$, or even $O(n + n^2 + 23)$, but just $\mathbf{O(n^2)}$

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)**

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: 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 ;
}
```

$\xrightarrow{}$ O(n)

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 ← 0 ;
    while (n > 0)
        count ← count + n mod 2;
        n ← 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
```

$\longrightarrow$ O(?)

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

$\longrightarrow$ 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++
```

$\longrightarrow$ O(?)

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

$\longrightarrow$ O(?)

Determining Big Oh: Calling Functions

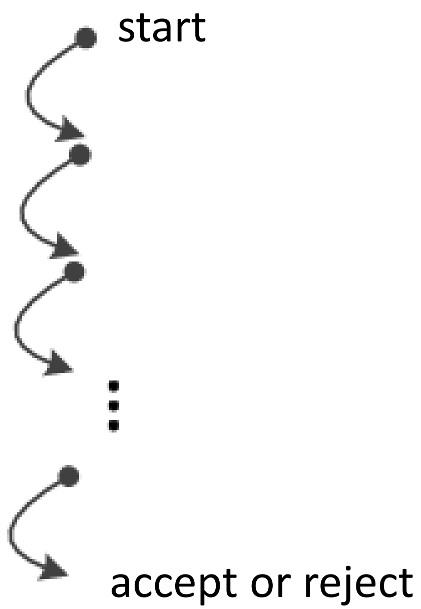
When one function calls another, big-Oh of calling function also considers big-Oh of called function and how many times embedded function(s) are called

```
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(?)
```

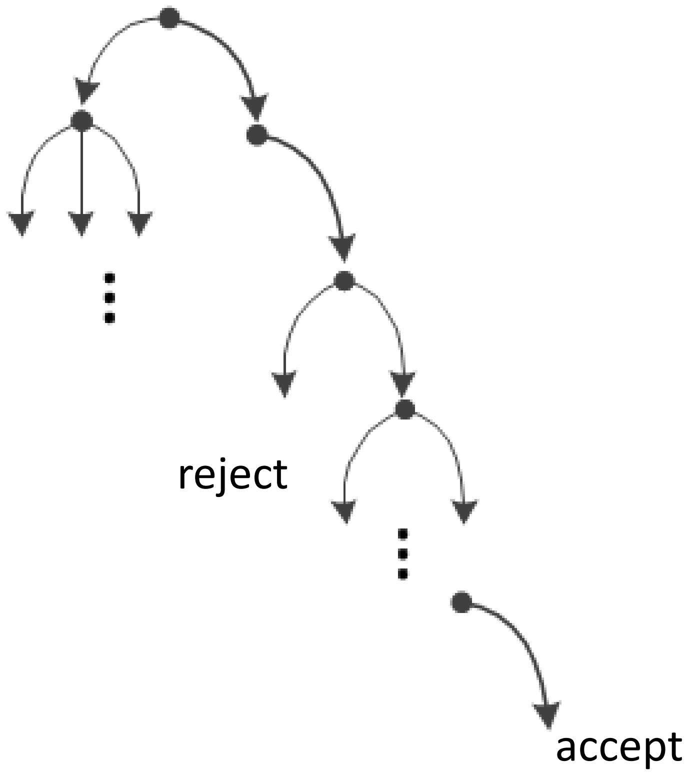
Non-deterministic Algorithms

Deterministic vs. non-deterministic algorithms

Deterministic



Non-deterministic



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 is a step that involves parallel processing
  - search a value in a vector
  - ordering the values of a vector
  - Determining if a number is prime

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: the best case occurs when x is present at the first location $\rightarrow$ T(n) = O(1)

Average case: We must know (or predict) the distribution of cases. Let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1)

$$Average\ time = \frac{\sum_{i=1}^{n+1} O(i)}{n+1} = \frac{O((n+1)\times(n+2)/2)}{n+1} = O(n)$$

Worst case: happens when the element to be searched is not present in the array $\rightarrow$ 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	2 bytes
	10 bytes

$S(n) = O(1)$

Big Oh: Recursion

For recursion it is necessary to determine:

– how many times will function 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) =

Torres de Hanói

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

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

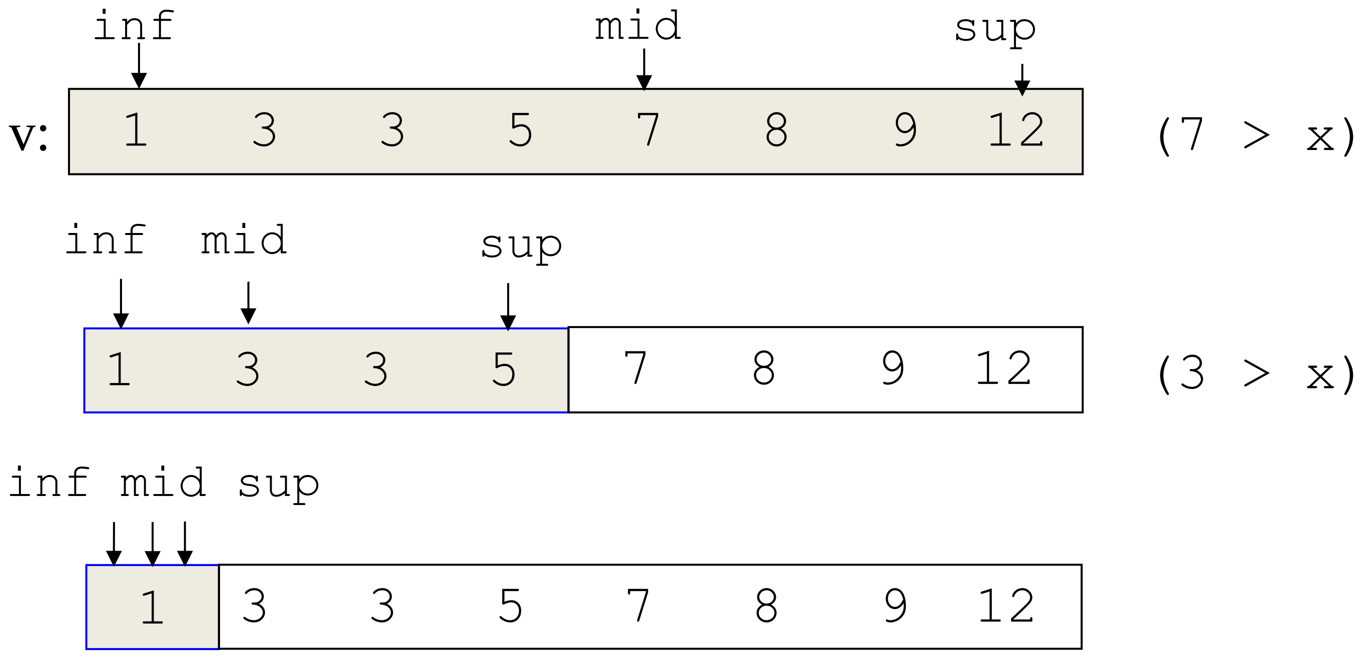
$T(n) = O(2^n)$ Exponencial

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 ½ until 1 element remains

  N, N/2, N/4, N/8, ..., 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, ..., N/4, N/2, N

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

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

Sorting

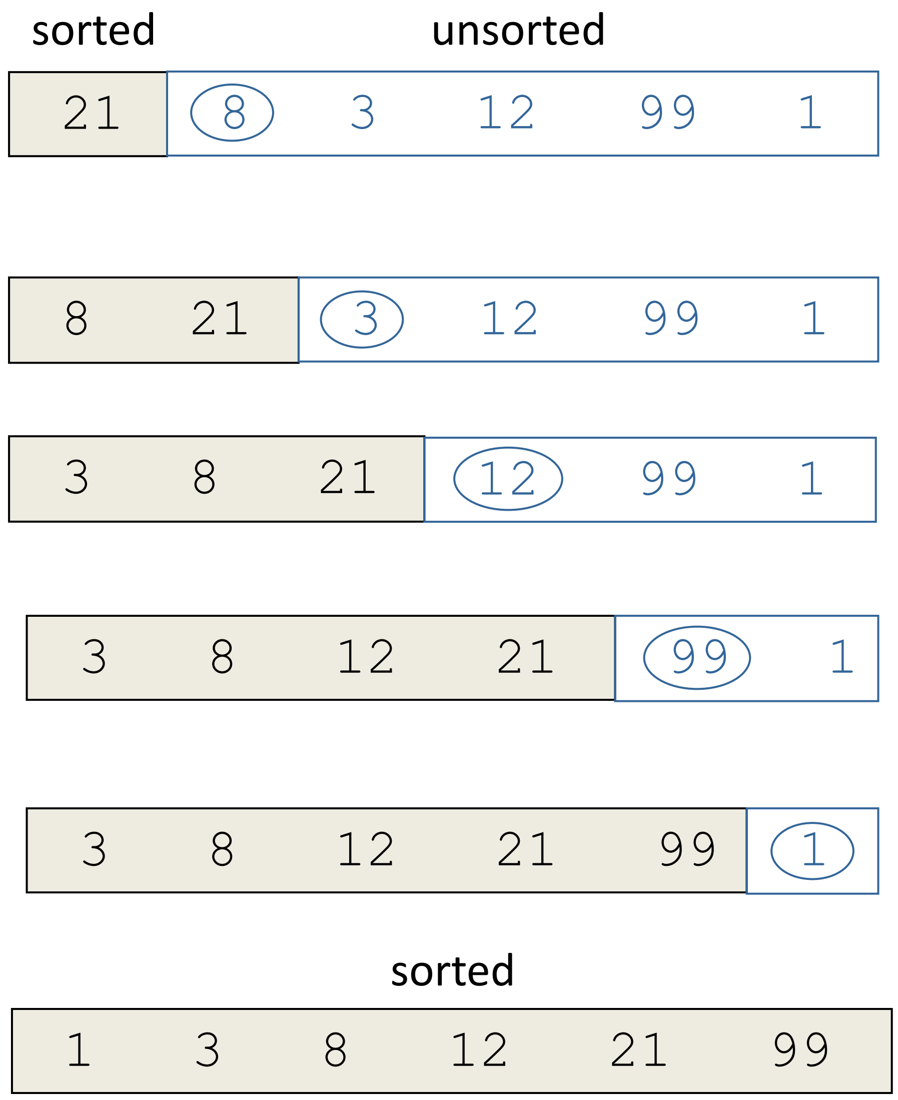
Basic problem: order elements in a vector

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

Insertion Sort



Insertion Sort

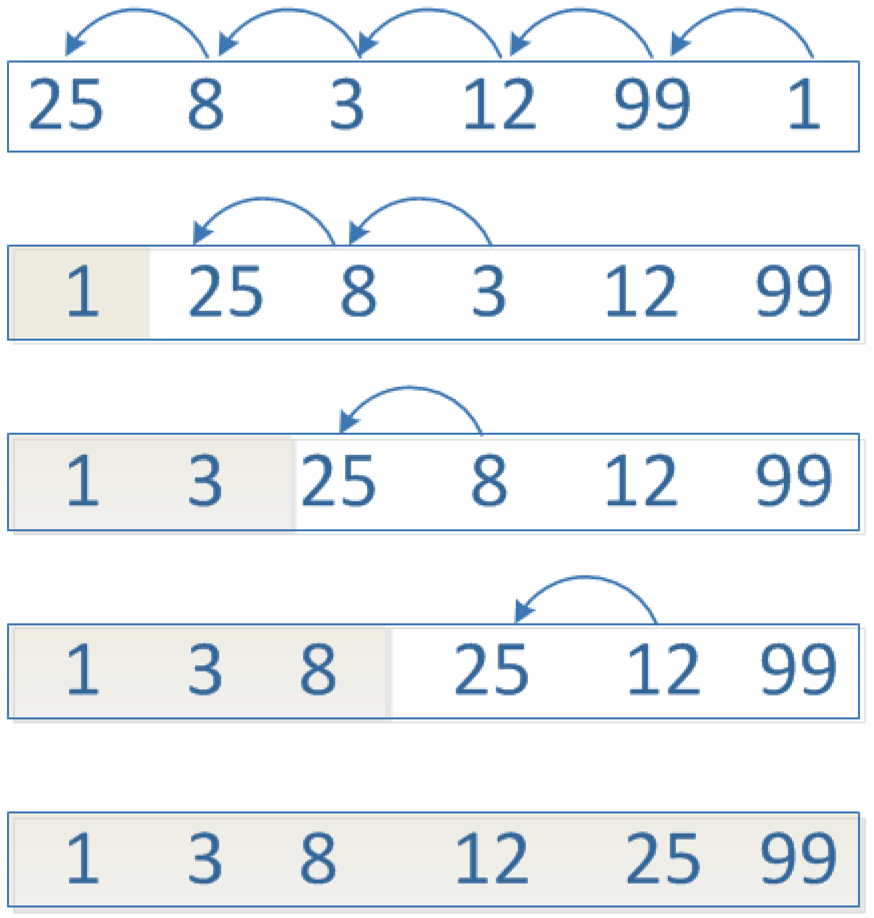
```
Algorithm void insertionSort (T v[], int n) {
    for (i ← 1 ; i < n ; i++) {
        j ← i
        x ← v[i]
        while(j > 0 && x < v[j-1]){
            v[j] ← v[j-1]
            j ← j-1
        }
        v[j] ← x ;
    }
}
```

Sub problem: Insertion of an element in a sorted vector

Best case: O(?)

Worst case: O(?)

Bubble Sort



1st iteration

2nd iteration

3th iteration

4th iteration

5th iteration

BubbleSort

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

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

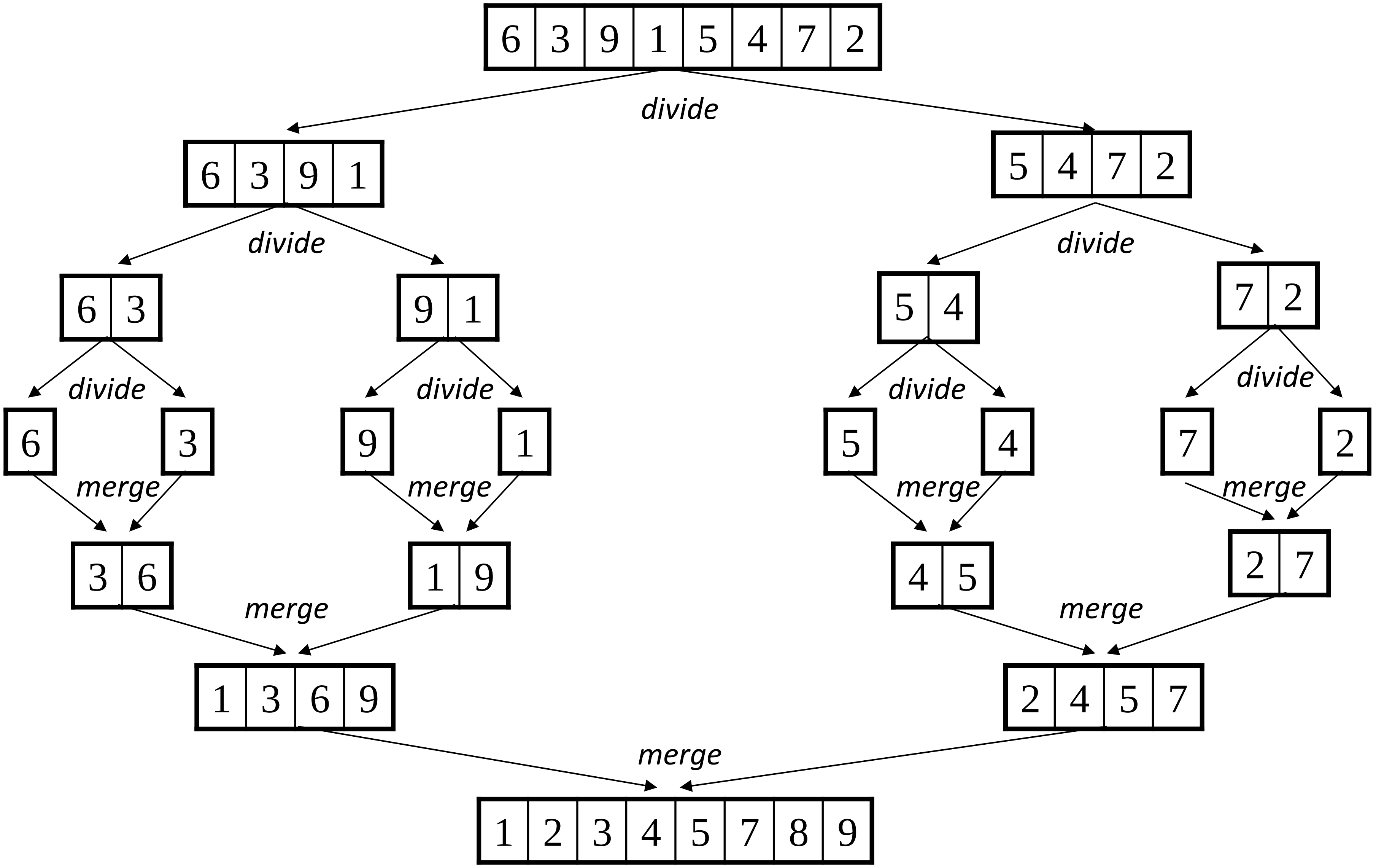
Merge-Sort Algorithm

Input: Sequence S with n elements

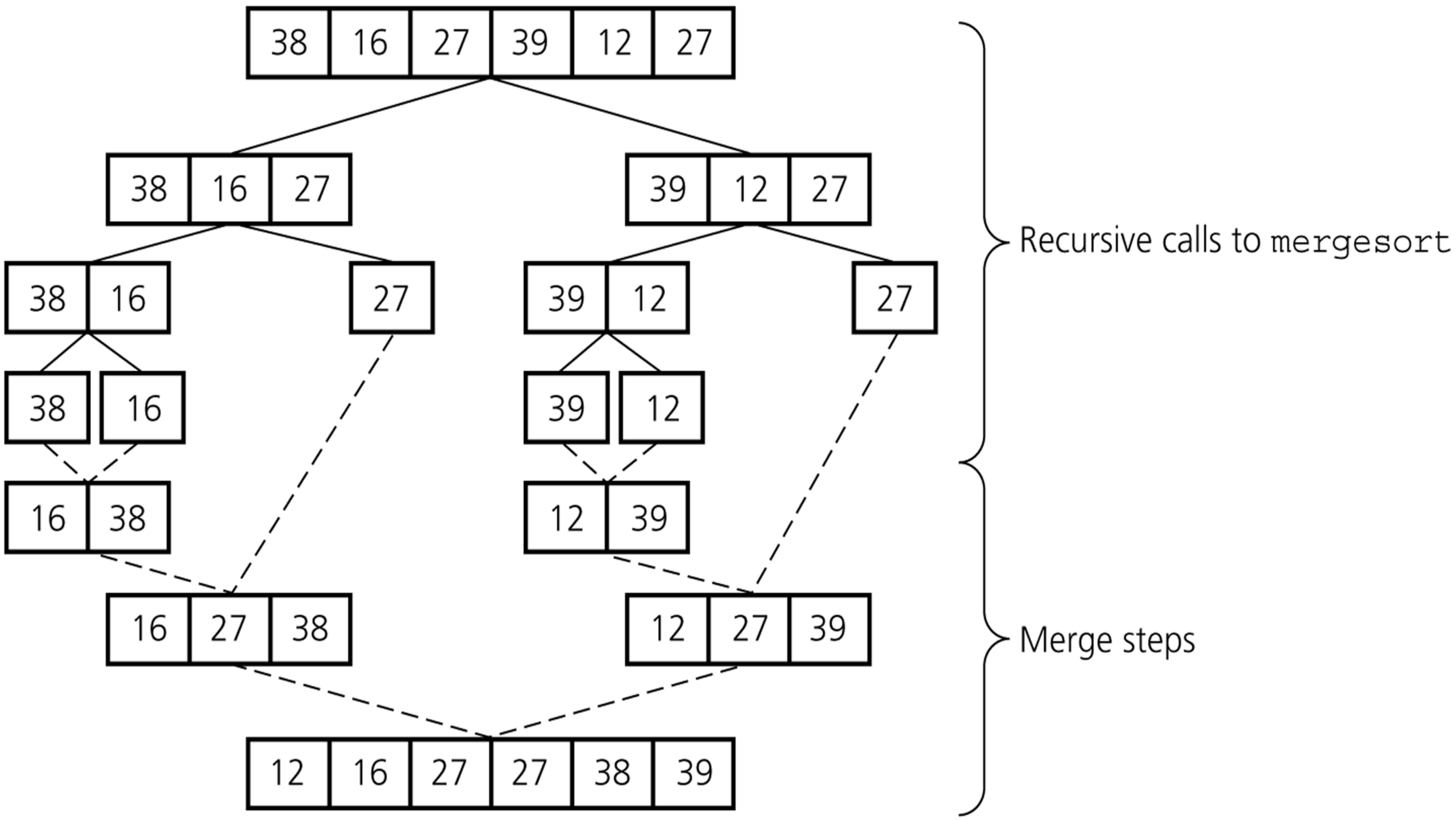
Output: Sequence S sorted

```
Algorithm void mergeSort (T S[]) {
   int n = S.length;
   if (n >= 2) {
      int mid = n/2; ;         // index of midpoint
      S1 <- S[0,..,mid]);
      S2 <- S[mid+1,..,n-1]);
      mergeSort(S1);
      mergeSort(S2);
     merge(S1, S2, S)    //merge two sorted sequences
}
```

Merge-Sort example

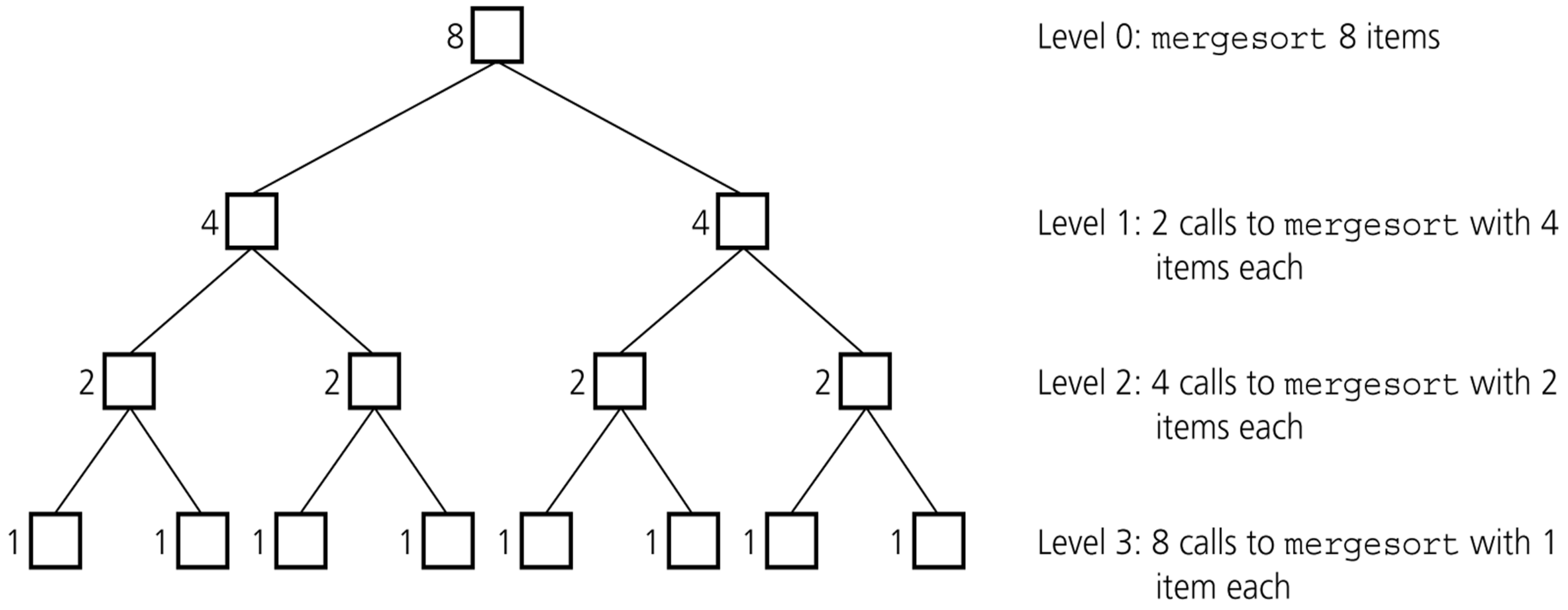


Merge-Sort another example



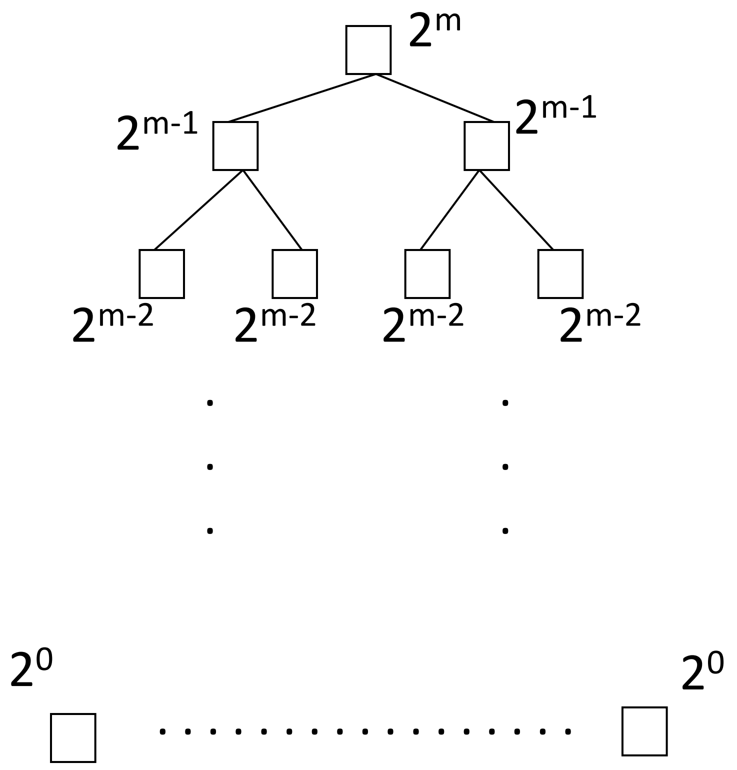
Analysis of Merge-Sort

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


8
4
4
2
2
2
2
1
1
1
1
1
1
1
1
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

Number of merges in each level



level 0 : 1 merge (size 2^{m-1})

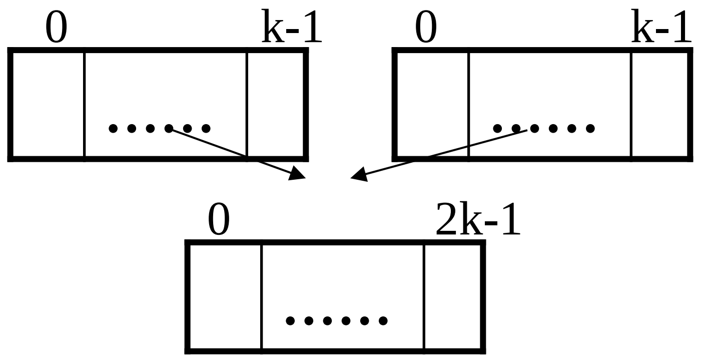
level 1 : 2 merges (size 2^{m-2})

level 2 : 4 merges (size 2^{m-3})

level m-1 : 2^{m-1} merges (size 2^0)

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 (nlogn)**

- Merge sort requires additional memory with size N

- Usually Merge-Sort is not used for main memory sorting, only for external memory sorting

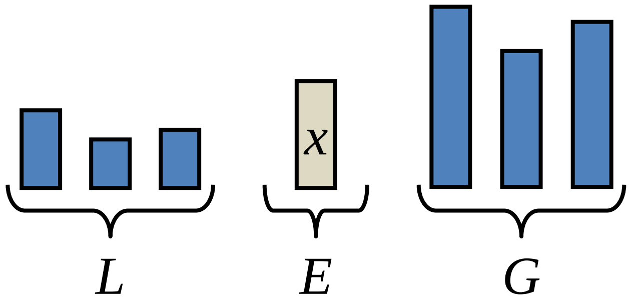
QuickSort Algorithm

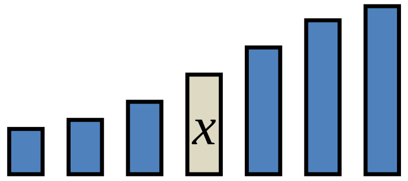
Like Merge-Sort, Quicksort is also based on the ***divide-and-conquer*** **paradigm**

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

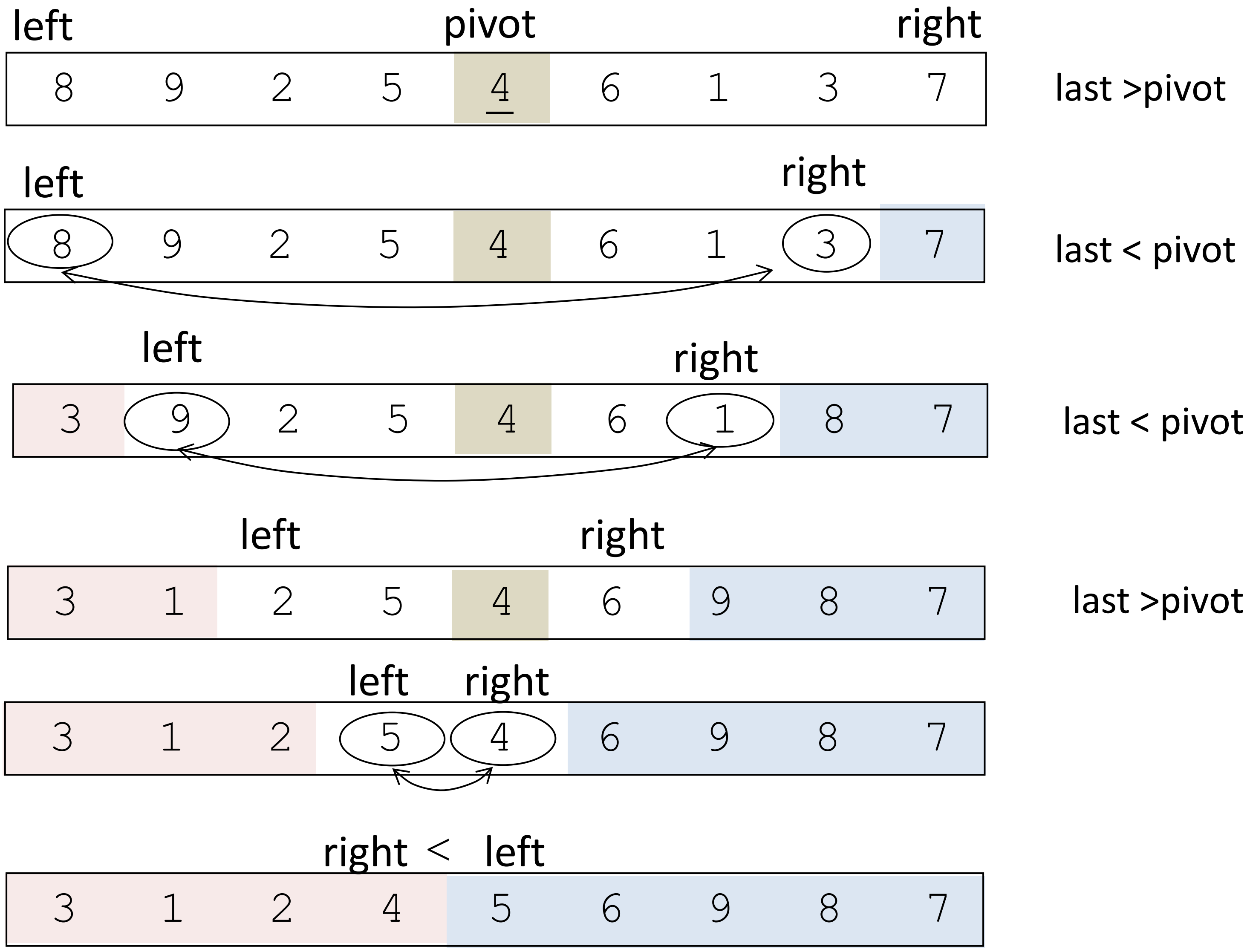
- Pick one element in the array, which will be the **pivot**

- 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 example



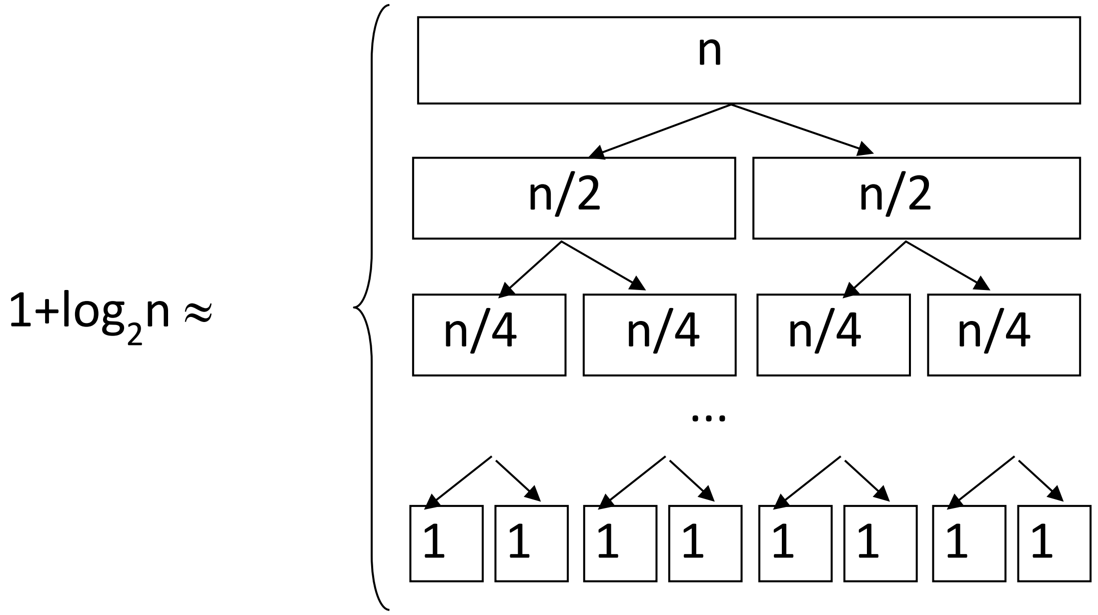
QuickSort Algorithm

```
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){
           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



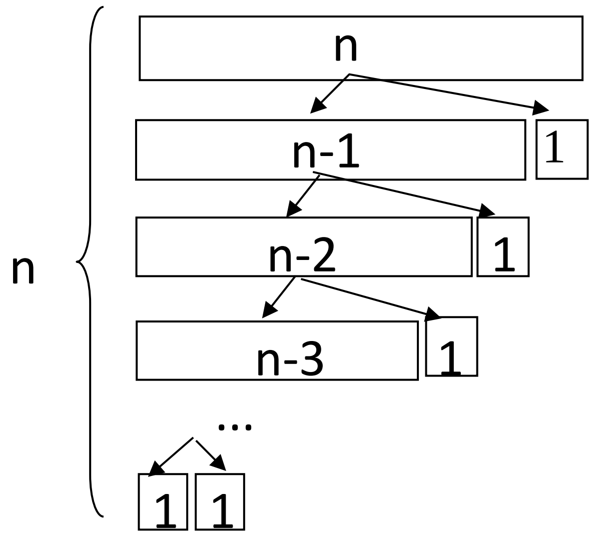
- There are **logN** partitions, and to obtain each partitions we do **N** comparisons (and not more than **N/2** swaps)
- Hence the complexity is **O(nlogn)**

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 $= n-1 + n-2 + \ldots + 1$

$= n^2/2 - n/2 \quad \rightarrow \quad \mathbf{O(n^2)}$

The number of swaps $= n-1 + n-1 + n-2 + \ldots + 1 \quad \rightarrow \quad \mathbf{O(n^2)}$

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

Choosing the Pivot

- The choice of the **pivot** influences the complexity of the algorithm
- If the pivot is the **smallest (or the largest)** element, the algorithm has its **worst complexity**

Choosing the Pivot

- **Some fixed element:** e.g. the first, the last, the one in the middle
  - Bad choice - may turn to be the smallest or the largest element, then one of the partitions will be empty
  - Randomly chosen (by random generator) - still a bad choice
- **The median of the array:**
  - if the array has N numbers, the median is the [N/2] largest number. This is difficult to compute - increases the complexity
- **The median-of-three choice:**
  - take the first, the last and the middle element
  - Choose the median of these three elements

Finding the Pivot

```
 Algorithm T median3 (T[] S, int left, int right){
     center ← (left + right)/2;
     if (S[left] < S[center])
         swap(S, left, center);
     if (S[center] > S[right])
         swap(S, center, right);
     if (S[left] > S[center])
         swap(S, left, center);
     swap(S,center,right-1);
     return S[right-1];
}
```

Analysis of QuickSort

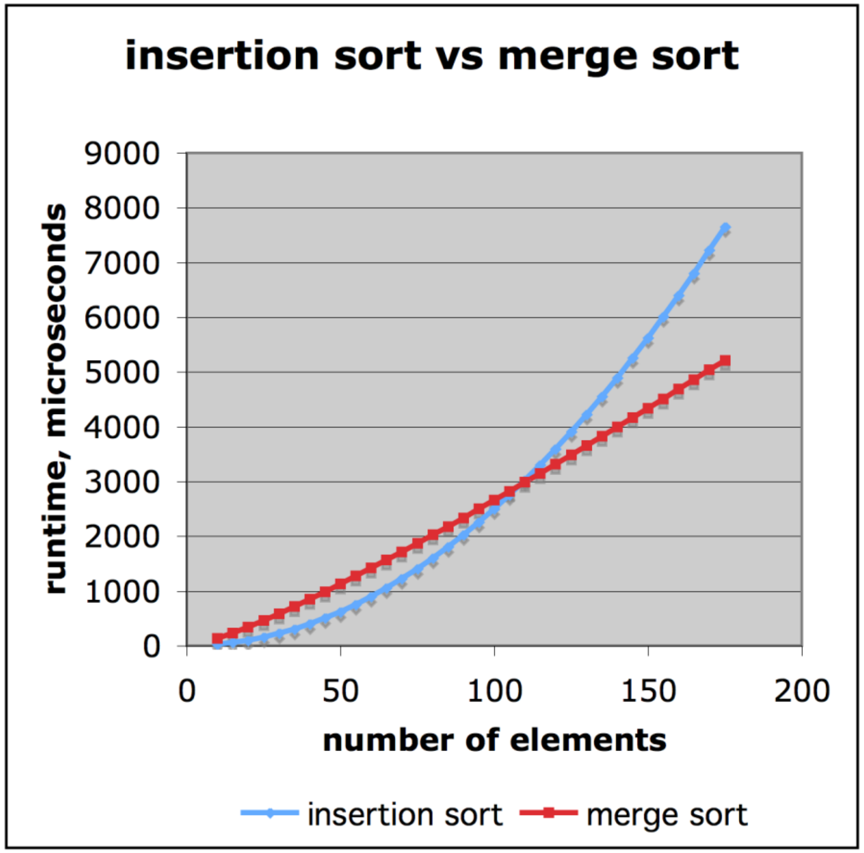
- One of the fastest algorithms on average O(nlogn)
- Does not need additional memory (the sorting takes place in the array - this is called in-place processing)
- The above advantages compensate for the rare occasions when it runs with O(n^2)

Faster computer vs. better Algorithm

insertion sort is $O(n^2)$

merge sort is O(nlogn)

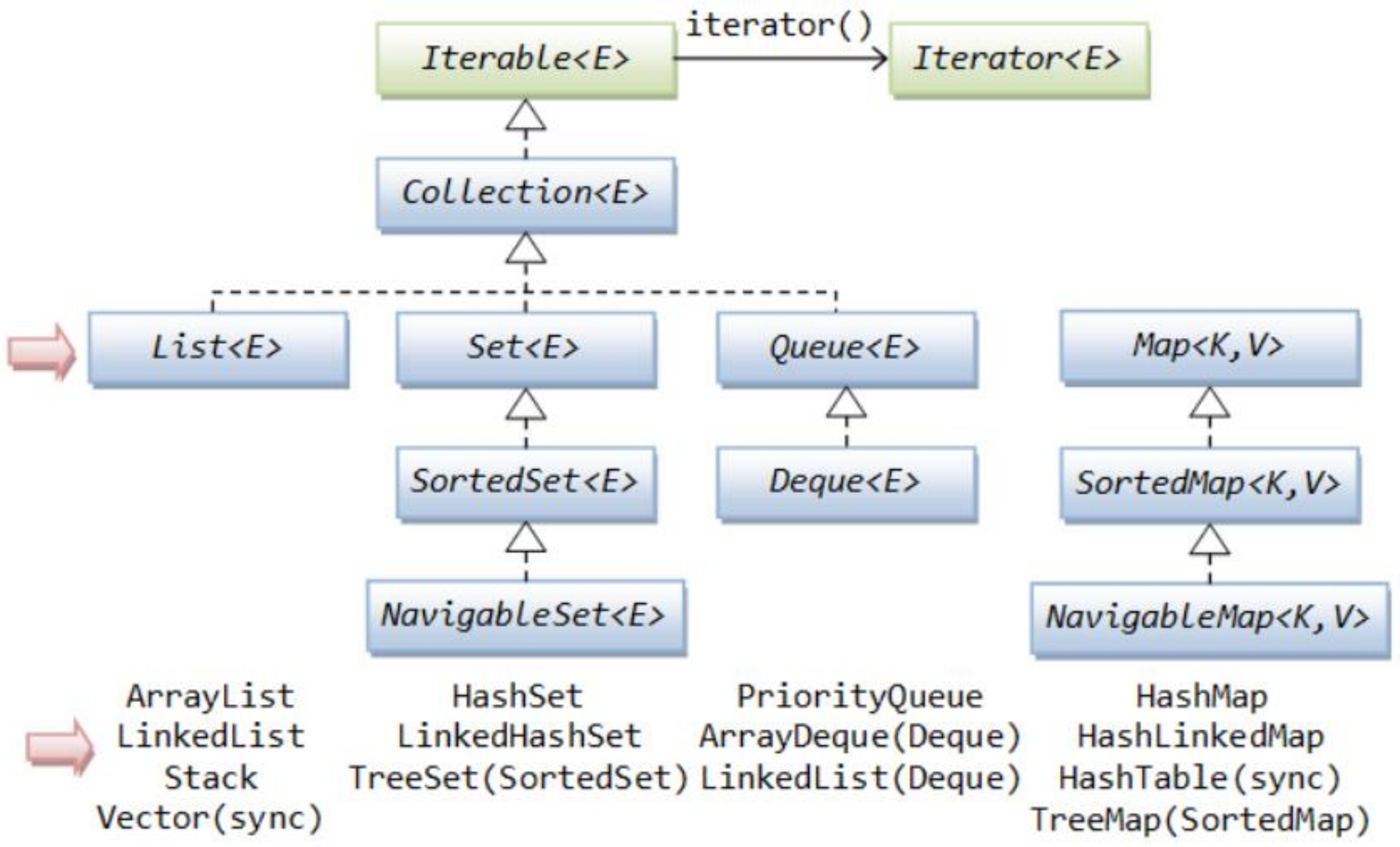
Sort a million items:
insertion sort takes roughly 70 hours
while
merge sort takes roughly 40 seconds



Algorithmic improvement is always more clever than hardware improvement

Complexity Analysis
some implementation classes
JAVA Collection Framework

The Collection Interfaces



List

A ***list*** is a dynamic collection of homogeneous elements A_0, A_1, …,A_{n-1} where A_i is the i-th element of the list. Positions range from 0 to n-1

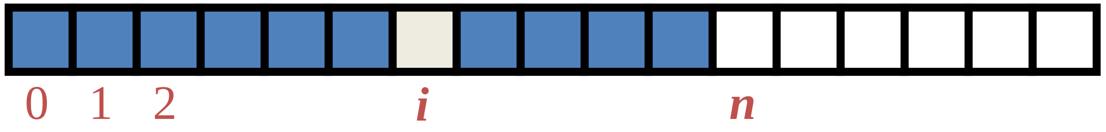
With a list it is possible:

- to store duplicate elements
- to specify where the element is stored
- to access the element by index

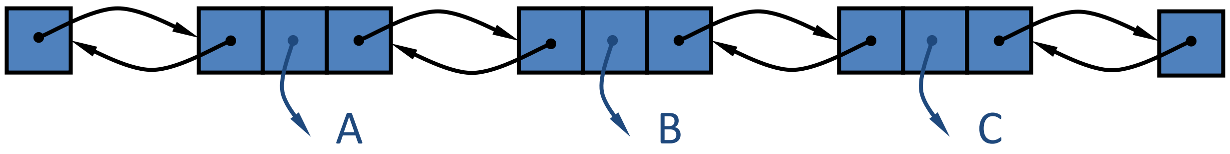
List<E> implementations

Two concrete implementations of the List ADT in the Java Collection:

- java.util.ArrayList



- java.util.LinkedList



Operations on ArrayList<E> and LinkedList<E>

Method	Complexity	
	ArrayList<E>	LinkedList<E>
`size()`	O(1)	O(1)
`isEmpty()`	O(1)	O(1)
`get(i)`	O(1)	O(min{i, n−i})
`set(i, e)`	O(1)	O(min{i, n−i})
`add(i, e)`	O(n)	O(min{i, n−i})
`add(0, e)`	O(n)	
`addFirst(e)`		O(1)
`add(e)`	O(1)	
`addLast(e)`		O(1)
`remove(i)`	O(n)	O(min{i, n−i})

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

Stack ADT

A ***stack*** is a collection of objects that are inserted and removed according to the ***last-in, first-out*** (***LIFO***) principle

With a stack it is possible:

- only access or remove the most recently inserted object that remains at the "top" of the stack

- **java.util.Stack** implements the LIFO semantics of a stack
- However, Java's Stack class remains only for historic reasons, and its interface is not consistent with most other data structures in the Java library
- It is recommended to use **double-ended queue** - a more general data structure that provides the LIFO functionality (and more)

Double-ended queue or *deque*

A ***queue*** is a collection whose elements are added and removed in a specific order, typically in a **first-in-first-out (FIFO)** manner

A ***deque*** is a double-ended queue that elements can be inserted and removed at both ends (head and tail) of the queue

A Deque can be used:

- **as FIFO queue** via methods:
  - add(e)/offer(e), remove()/poll(), element()/peek()
- **as LIFO queue** via methods:
  - push(e), pop(), peek()

Queue and Deque implementations

- **PriorityQueue<E>**: A queue implemented with a heap where the elements are ordered based on an ordering specified, instead of FIFO

- **ArrayDeque<E>**: A queue and deque implemented based on a circular array

- **LinkedList<E>**: The LinkedList<E> also implements the Queue<E> and Deque<E> interfaces, in addition to the List<E> interface, providing a queue or deque that is implemented as a double-linked list data structure

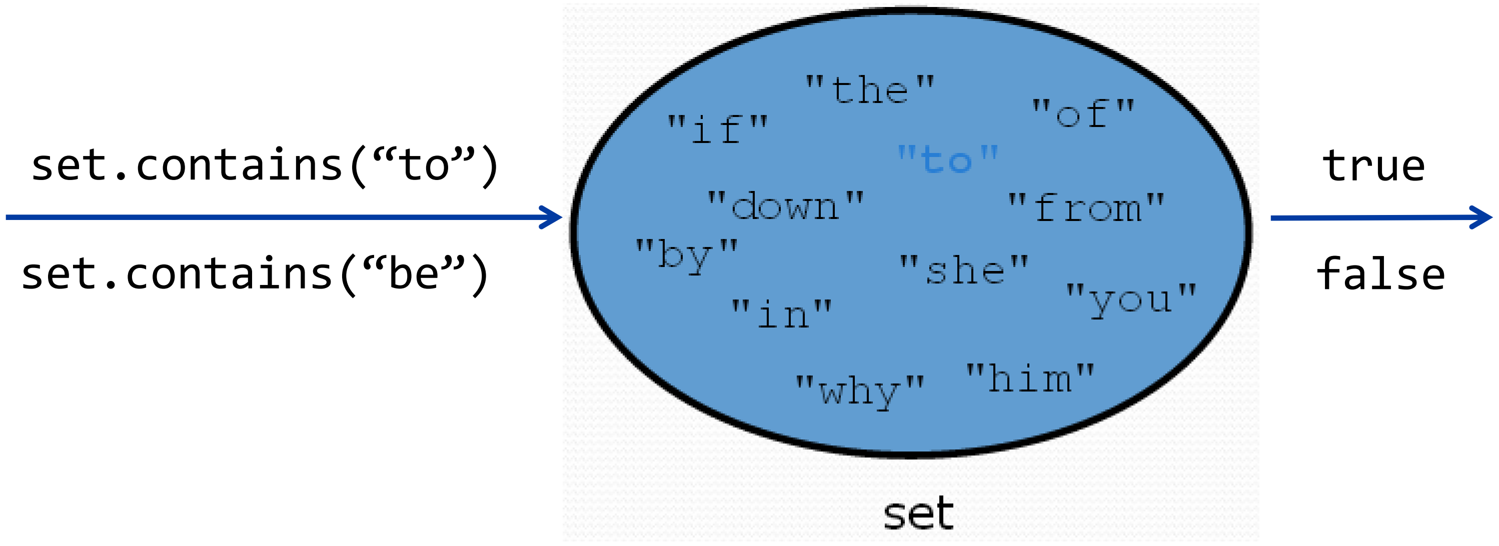
→ Complexity analysis of operations?

Set

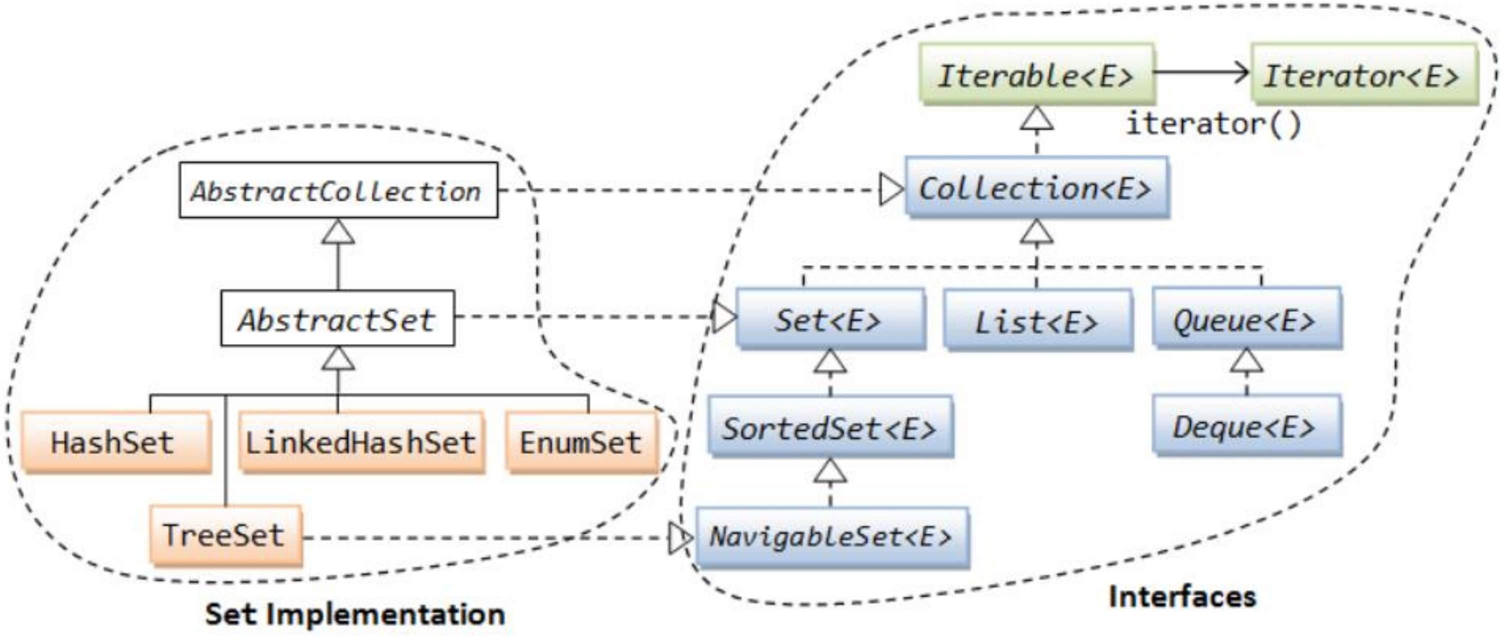
Set is a collection of unique values (no duplicates allowed) that can perform the following operations efficiently:

- add, remove, search (contains)

We don't think of a set as having indexes; we just add things to the set in general and don't worry about order



Set<E> Interfaces



The Set<E> interface abstract methods:

```
boolean add(E o)              //add the specified element if it is not already present
boolean remove(Object o)  // remove the specified element if it is present
boolean contains(Object o)  // return true if it contains o

// Set operations
boolean addAll(Collection<? extends E> c)       //Set union
boolean retainAll(Collection<?> c)              //Set intersection
```

Set<E> Implementations

- In Java, sets are represented by Set type in java.util

Set is implemented by:

- **HashSet:** implemented using a "hash table" array
  - very fast: O(1) for all operations
  - elements are stored in unpredictable order

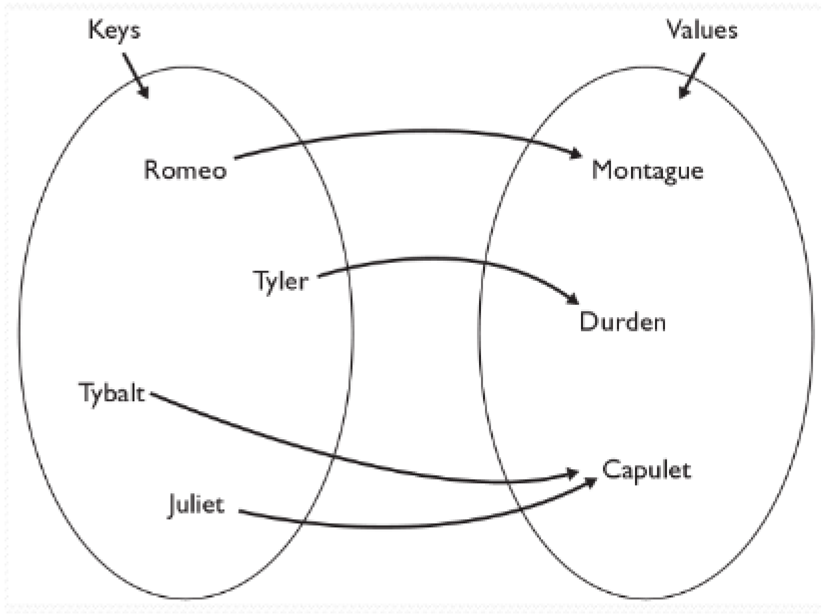
- **TreeSet:** implemented using a "binary search tree“
  - pretty fast: O(log n) for all operations
  - elements are stored in sorted order

- **LinkedHashSet:** stores the elements in a linked-list hash table
  - very fast: O(1) for all operations
  - stores in order of insertion

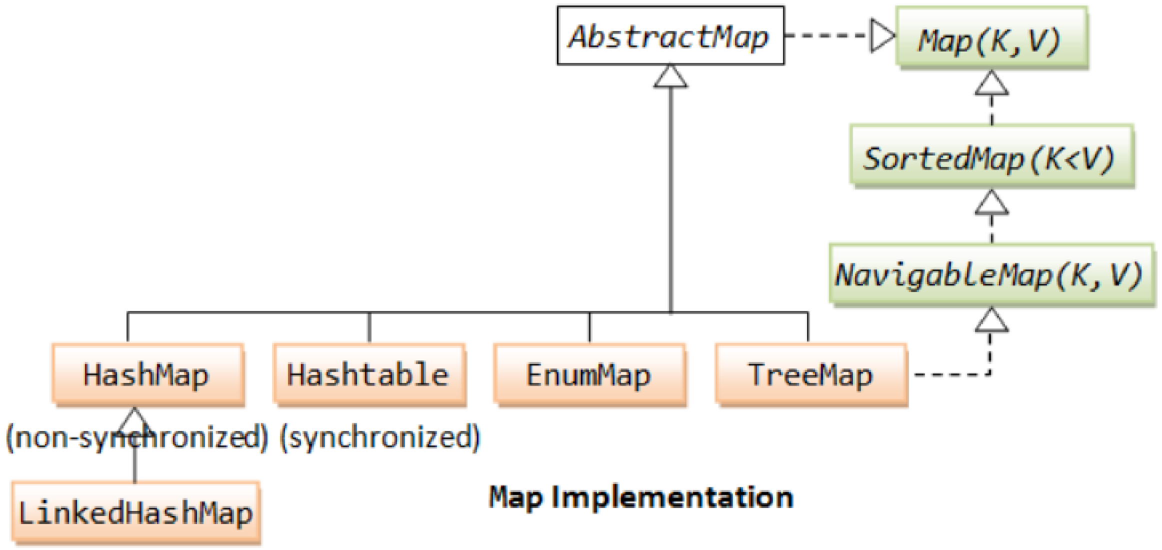
Map

- A **map** is a collection of key-value pairs. Each key maps to one and only value. Duplicate keys are not allowed, but duplicate values are allowed

- Maps are similar to linear arrays, except that an array uses an integer key to index and access its elements; whereas a map uses any arbitrary key (such as Strings or any objects)



Map<K,V> Interfaces



Map Implementation

Map<K,V> interface abstract methods:

```
V get(Object key)             //Returns the value of the specified key
put(K key, V value)           //Associates the specified value with the specified key
boolean containsKey(Object key)
boolean containsValue(Object value)

// Views
Set<K> keySet()          // Returns a set view of the keys Collection<V>
values()                 // Returns a collection view of the values Set
entrySet()               // Returns a set view of the key-value
```

Map<K,V> Implementations

in Java, maps are represented by Map type in java.util

Map is implemented by:

- **HashMap:** implemented using an array called a "hash table“
  - extremely fast: O(1)
  - keys are stored in unpredictable order

- **TreeMap:** implemented as a linked "binary tree" structure
  - very fast: O(log N)
  - keys are stored in sorted order

- **LinkedHashMap**
  - extremely fast: O(1)
  - keys are stored in order of insertion