# COMP 251: Recurrences

Jérôme Waldispühl & Giulia Alberini
School of Computer Science
McGill University

Based on slides from Hatami, Bailey, Stepp & Martin, Snoeyink.

# Outline

- Introduction: Thinking recursively
- Definition
- Examples:
    - Binary search
    - Fibonacci numbers
    - Merge sort (bonus: insertion sort)
    - Quicksort
- Running time
- Substitution method

# Course credits

c(x) = total number of credits required to complete course x

c(COMP462) = ?

= 3 credits **+ #credits for prerequisites**

COMP462 has 2 prerequisites: COMP251 & MATH323

$$= 3\text{ credits} + \underbrace{\mathbf{c(COMP251)}} + \underbrace{\mathbf{c(MATH323)}}$$

*The function c calls itself twice*

c(COMP251) = ?          c(MATH323) = ?

c(COMP251) = 3 credits + c(COMP250) COMP250 is a prerequisite

Substitute c(COMP251) into the formula:

c(COMP462) = 3 credits + 3 credits + c(COMP250) + c(MATH323)

c(COMP462) = 6 credits + c(COMP250) + c(MATH323)

# Course credits

c(COMP462) = 6 credits + c(COMP250) + c(MATH323)

  c(COMP250) = ?    c(MATH323) = ?

  c(COMP250) = 3 credits # no prerequisite

c(COMP462) = 6 credits + 3 credits + c(MATH323)

  c(MATH240) = ?

  c(MATH240) = 3 credits + c(MATH141)

c(COMP462) = 9 credits + 3 credits + c(MATH141)

  c(MATH141) = ?

  c(MATH141) = 4 credits # no prerequisite

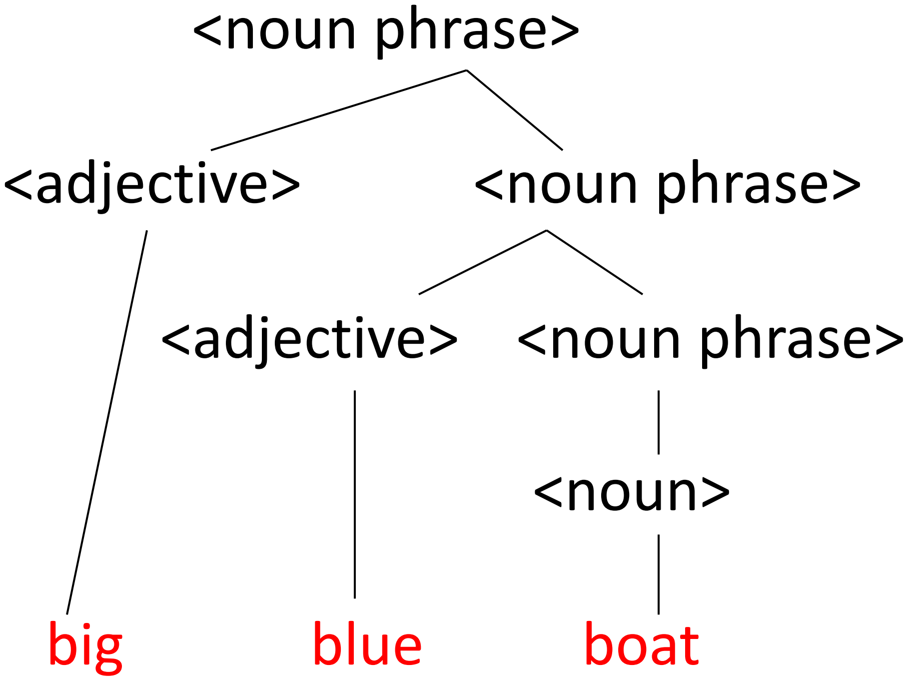
c(COMP462) = 12 credits + 4 credits = 16 credits

# Recursive definition

A noun phrase is either

- a noun, or
- an adjective followed by a noun phrase

<noun phrase> ⟶ <noun> OR <adjective> <noun phrase>



<noun phrase>

<adjective> <noun phrase>

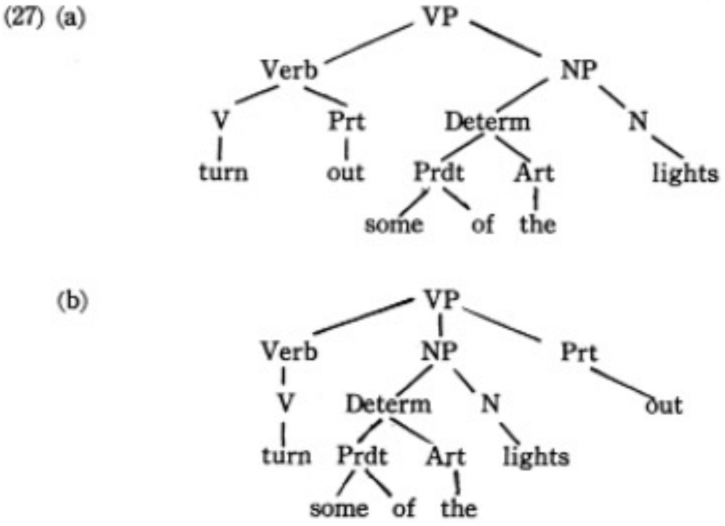
<adjective> <noun phrase>

<noun>

big blue boat

# Aside on grammars

- The previous slide was a simplified example of how we can use grammars to define sentences
- Grammars also exist outside of natural language.
- They are commonly used in computer science to represent as tree any kind of sequence of words, events, nucleotides
- Trees contains more information than the string alone.
- You can read more here: [https://en.wikipedia.org/wiki/Formal_grammar](https://en.wikipedia.org/wiki/Formal_grammar)
- You will discuss this in more detail in COMP 330!



**Chomsky, Noam**. "On the notion 'rule of grammar'." *Proceedings of the Twelfth Symposium in Applied Mathematics*. Vol. 12. American Mathematical Society, 1961.

# Definitions

**Recursive definition:**
A definition that is defined in terms of itself.

**Recursive method:**
A method that calls itself (directly or indirectly).

**Recursive programming:**
Writing methods that call themselves to solve problems recursively.

# Why using recursions?

- "cultural experience" - A different way of thinking of problems
- Can solve some kinds of problems better than iteration
- Leads to elegant, simplistic, short code (when used well)
- Many programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively  (no loops)
- Recursions can be a good alternative to iteration (loops).

# Definition

**Definition (recurrence):**

A ***recurrence*** is a function that is defined in terms of

- one or more base cases, and
- itself, but with smaller arguments.

Examples:

$$T(n) = \begin{cases} 1 & if\ n = 1 \\ T(n-1) + 1 & if\ n > 1 \end{cases} \qquad T(n) = \begin{cases} 1 & if\ n = 1 \\ T\left(\frac{n}{3}\right) + T\left(\frac{2 \cdot n}{3}\right) + n & if\ n > 1 \end{cases}$$

Many technical issues:

- Floors and ceilings
- Exact vs. *asymptotic* functions
- Boundary conditions

Note: we will usually express the solution of the recurrence using the *asymptotic* notation.

# Iterative algorithms

**Definition (iterative algorithm):** Algorithm where a problem is solved by iterating (going step-by-step) through a set of commands, often using loops.

```
Algorithm: power(a,n)
Input: non-negative integers a, n
Output: $a^n$
product ← 1
for i = 1 to n do
        product ← product * a
return product
```

| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| product | 1 | a | $a * a = a^2$ | $a^2 * a = a^3$ | $a^3 * a = a^4$ |

# Recursive algorithms

**Definition (Recursive algorithm):** algorithm is recursive if in the process of solving the problem, it calls itself one or more times.

```
Algorithm: power(a,n)
Input: non-negative integers a, n
Output: $a^n$
if (n=0) then
   return 1
else
   return a * power(a,n-1)
```

# Example

power(7,4) calls

- ↳ 7 * power(7,3)
  - ↳ 7 * power(7,2)
    - ↳ 7 * power(7,1)
      - ↳ 7 * power(7,0)
        - ↳ 1
      - 7 * 1 = 7
    - 7 * 7 = 49
  - 7 * 49 = 343
- 7 * 343 = 2041

# Algorithm structure

Every recursive algorithm involves at least 2 cases:

> **base case**: A simple occurrence that can be answered directly.
>
> **recursive case**: A more complex occurrence of the problem that cannot be directly answered but can instead be described in terms of *smaller occurrences* of the same problem.

Some recursive algorithms have more than one base or recursive case, but all have at least one of each.

**A crucial part of recursive programming is identifying these cases.**

# Binary Search

**Algorithm binarySearch(array, start, stop, key)**
**Input:** - A **sorted** array
- the region [start , ... , stop] (inclusively) to be searched
- the key to be found

**Output:** returns the index at which the key has been found, or returns -1 if the key is not in array [start , ... , stop].

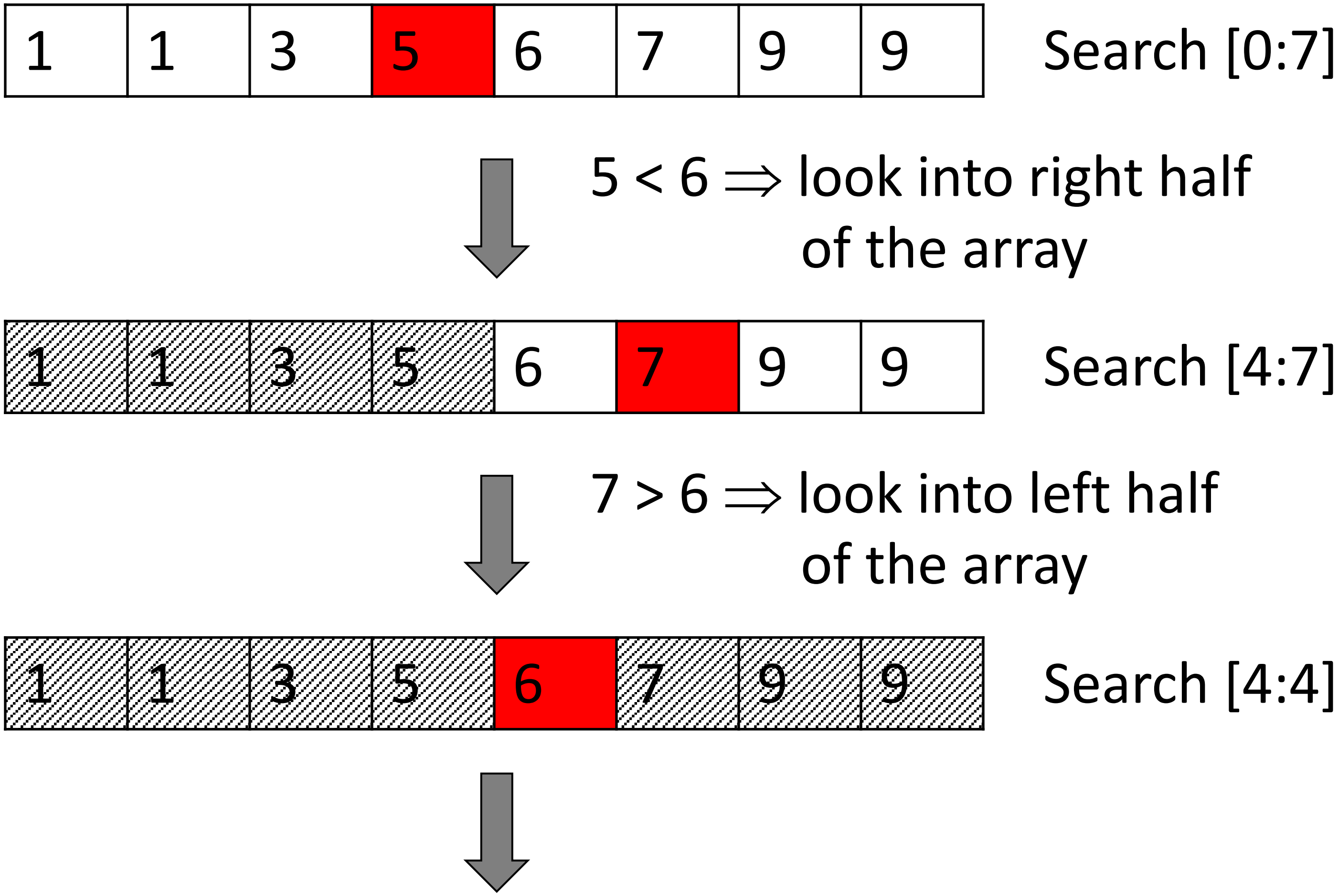
**Example:** Does the following **sorted** array A contains the number 6?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A = | 1 | 1 | 3 | 5 | 6 | 7 | 9 | 9 |

Call: `binarySearch(A, 0, 7, 6)`

# Binary search example

We are splitting the array in two at each step



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 5 | 6 | 7 | 9 | 9 |

Search [0:7]

5 < 6 ⇒ look into right half of the array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 5 | 6 | 7 | 9 | 9 |

Search [4:7]

7 > 6 ⇒ look into left half of the array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 5 | 6 | 7 | 9 | 9 |

Search [4:4]

6 is found. Return 4 (index)

# Binary Search Algorithm

```
int bsearch(int[] A, int i, int j, int x) {
   if (i<=j) { // the region to search is non-empty
      int e = ⌊(i+j)/2⌋;
      if (A[e] > x) {
         return bsearch(A,i,e-1,x);
      } else if (A[e] < x) {
         return bsearch(A,e+1,j,x);
      } else {
         return e;
      }
   } else { return -1; } // value not found
}
```

# Fibonacci numbers

$Fib_0 = 0$ base case

$Fib_1 = 1$ base case

$Fib_n = Fib_{n-1} + Fib_{n-2}$ for $n > 1$ recursive case

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $Fib_i$ | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 |

# Recursive algorithm

Compute Fibonacci number n (for n ≥ 0)

```
public static int Fib(int n) {
   if (n <= 1) {
      return n;
   }
   // {n > 1}
   return Fib(n-1) + Fib(n-2);
}
```

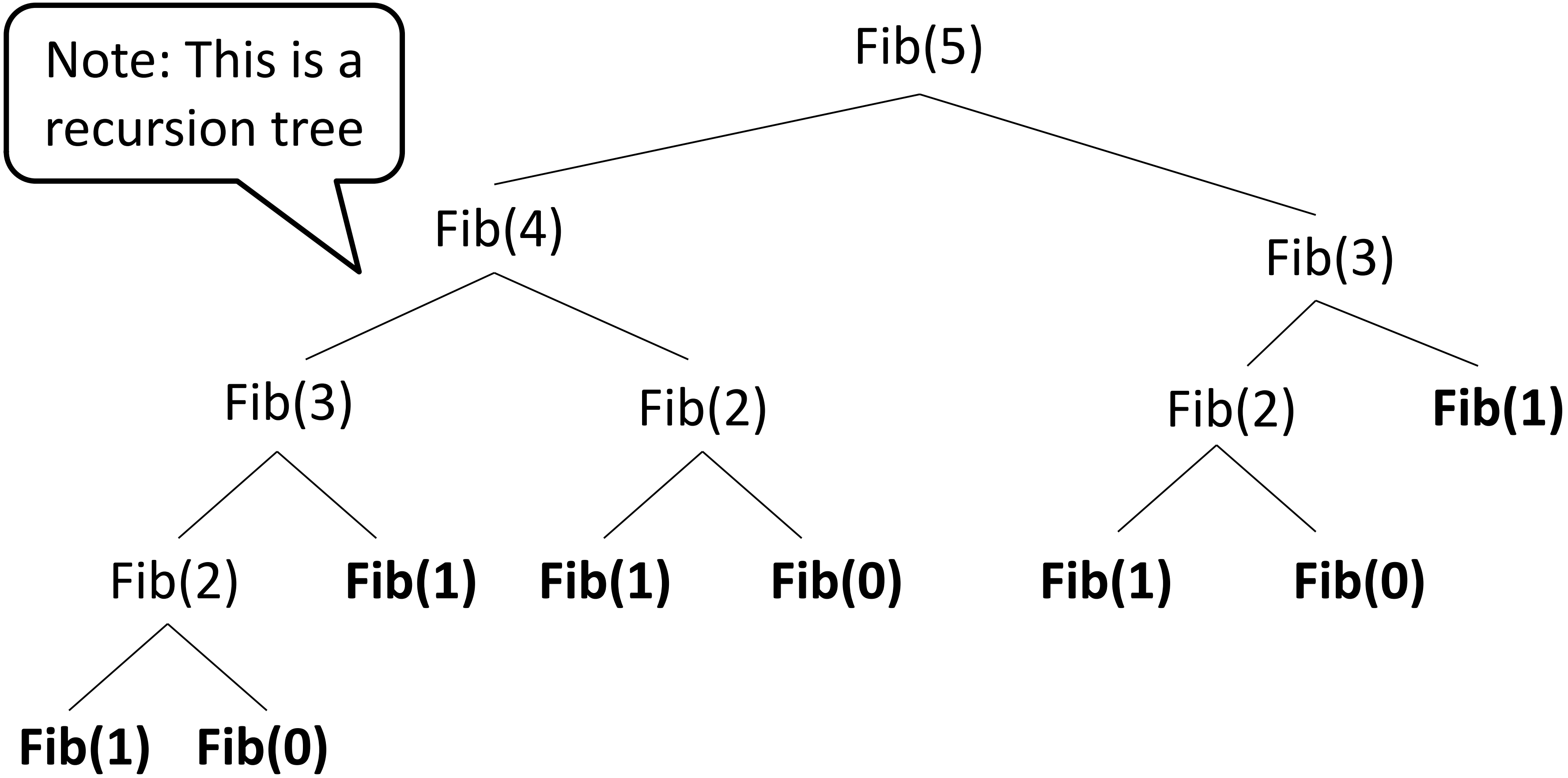
Can handle both base cases together

Recursive case (2 recursive calls)

**Note:** The algorithm follows the definition of Fibonacci numbers.

# Recursion is not always efficient!

Note: This is a recursion tree



Fib(5)

Fib(4) Fib(3)

Fib(3) Fib(2) Fib(2) **Fib(1)**

Fib(2) **Fib(1)** **Fib(1)** **Fib(0)** **Fib(1)** **Fib(0)**

**Fib(1)** **Fib(0)**

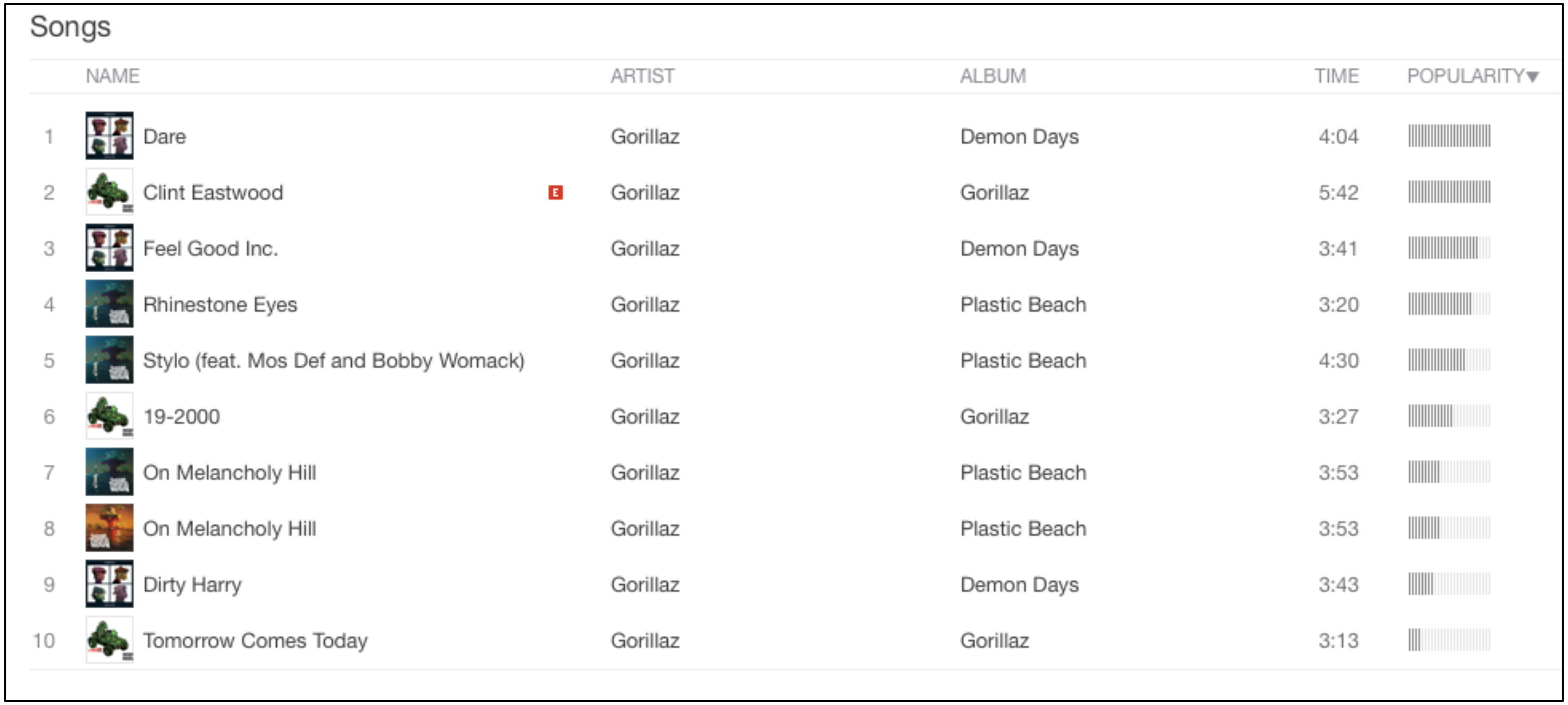
Question: When computing Fib(n), how many times are Fib(0) or Fib(1) called?

# Designing recursive algorithms

- To write a recursive algorithm:
  - Find how the problem can be broken up in one or more smaller problems of the same nature
  - Remember the base case!
- Naive implementation of recursive algorithms may lead to prohibitive running time.
  - Naive Fibonacci $\Rightarrow$ $O(\phi^n)$ operations
  - Better Fibonacci $\Rightarrow$ $O(\log n)$ operations
- Usually, better running times are obtained when the size of the sub-problems are approximately equal.
  - power(a,n) = a * power(a,n-1) $\Rightarrow$ O(n) operations
  - power(a,n) = $(\text{power}(a,n/2))^2$ $\Rightarrow$ O(log n) operations

# Sorting problem

**Problem:** Given a list of *n* elements from a totally ordered universe, rearrange them in ascending order.



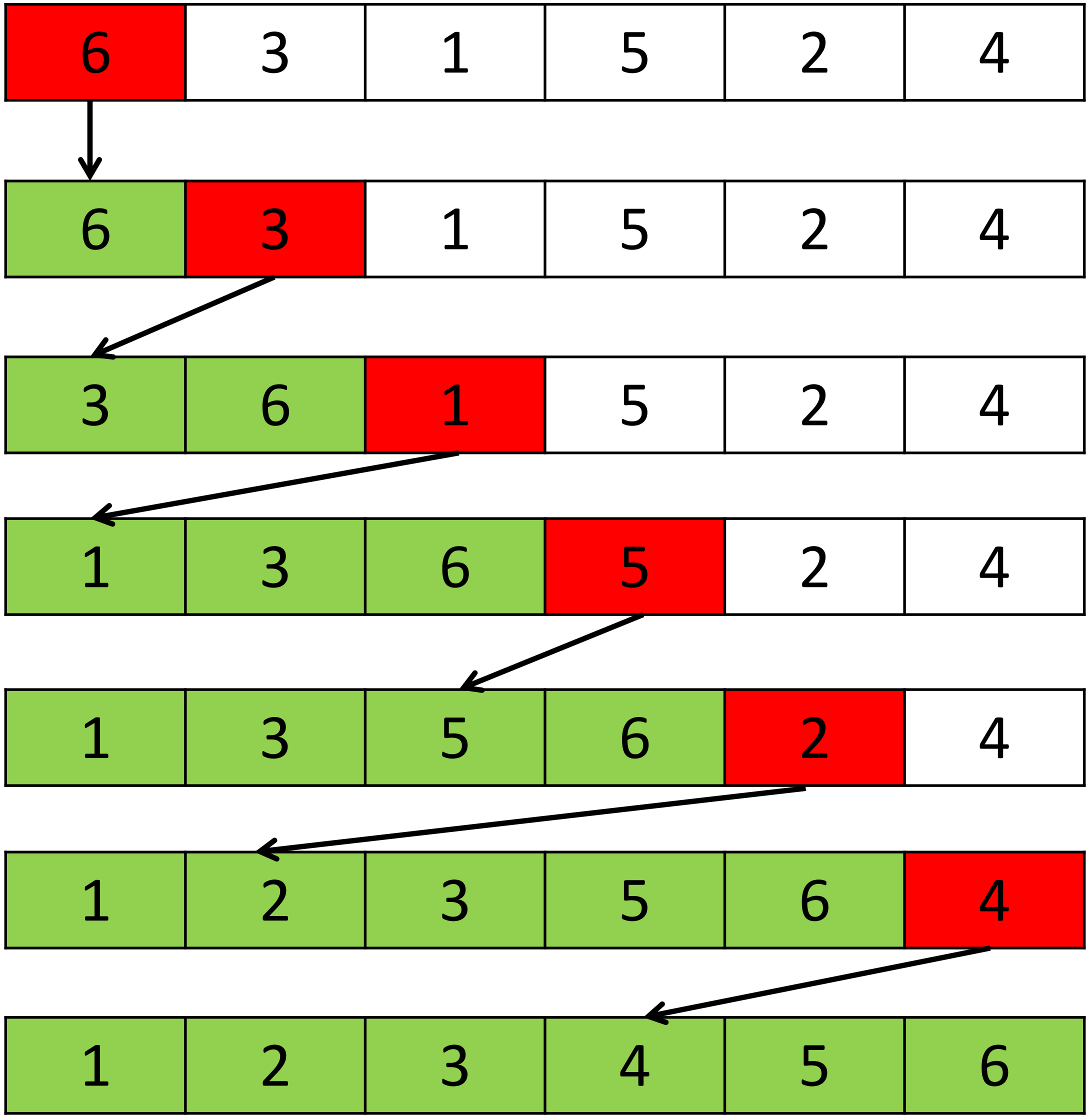
Songs

| | NAME | | ARTIST | ALBUM | TIME | POPULARITY▼ |
|---|---|---|---|---|---|---|
| 1 | Dare | | Gorillaz | Demon Days | 4:04 | |
| 2 | Clint Eastwood | E | Gorillaz | Gorillaz | 5:42 | |
| 3 | Feel Good Inc. | | Gorillaz | Demon Days | 3:41 | |
| 4 | Rhinestone Eyes | | Gorillaz | Plastic Beach | 3:20 | |
| 5 | Stylo (feat. Mos Def and Bobby Womack) | | Gorillaz | Plastic Beach | 4:30 | |
| 6 | 19-2000 | | Gorillaz | Gorillaz | 3:27 | |
| 7 | On Melancholy Hill | | Gorillaz | Plastic Beach | 3:53 | |
| 8 | On Melancholy Hill | | Gorillaz | Plastic Beach | 3:53 | |
| 9 | Dirty Harry | | Gorillaz | Demon Days | 3:43 | |
| 10 | Tomorrow Comes Today | | Gorillaz | Gorillaz | 3:13 | |

Classical problem in computer science with many different algorithms (bubble sort, merge sort, quick sort, etc.)

# Insertion sort (example)

We are expanding a sorted region by traversing the list from left to right and swapping from right to left until the key is at a right place.



6 3 1 5 2 4

6 3 1 5 2 4

3 6 1 5 2 4

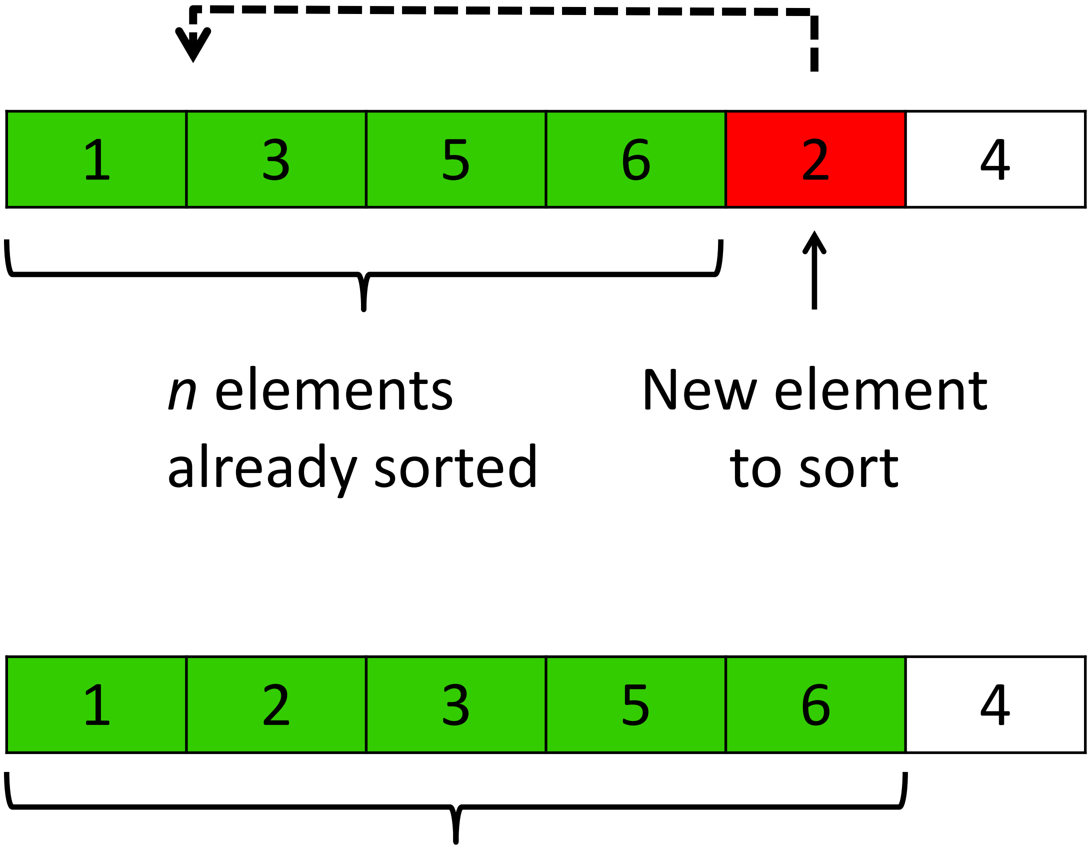
1 3 6 5 2 4

1 3 5 6 2 4

1 2 3 5 6 4

1 2 3 4 5 6

# Insertion sort (principle)



| | | | | | |
|---|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 2 | 4 |

*n* elements
already sorted

New element
to sort

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 6 | 4 |

*n+1* elements sorted

# Insertion sort (iterative algorithm)

```
For i ← 1 to length(A) - 1
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
end for
```

- Iterative method to sort objects.
- Relatively slow, we can do better using a *recursive* approach!

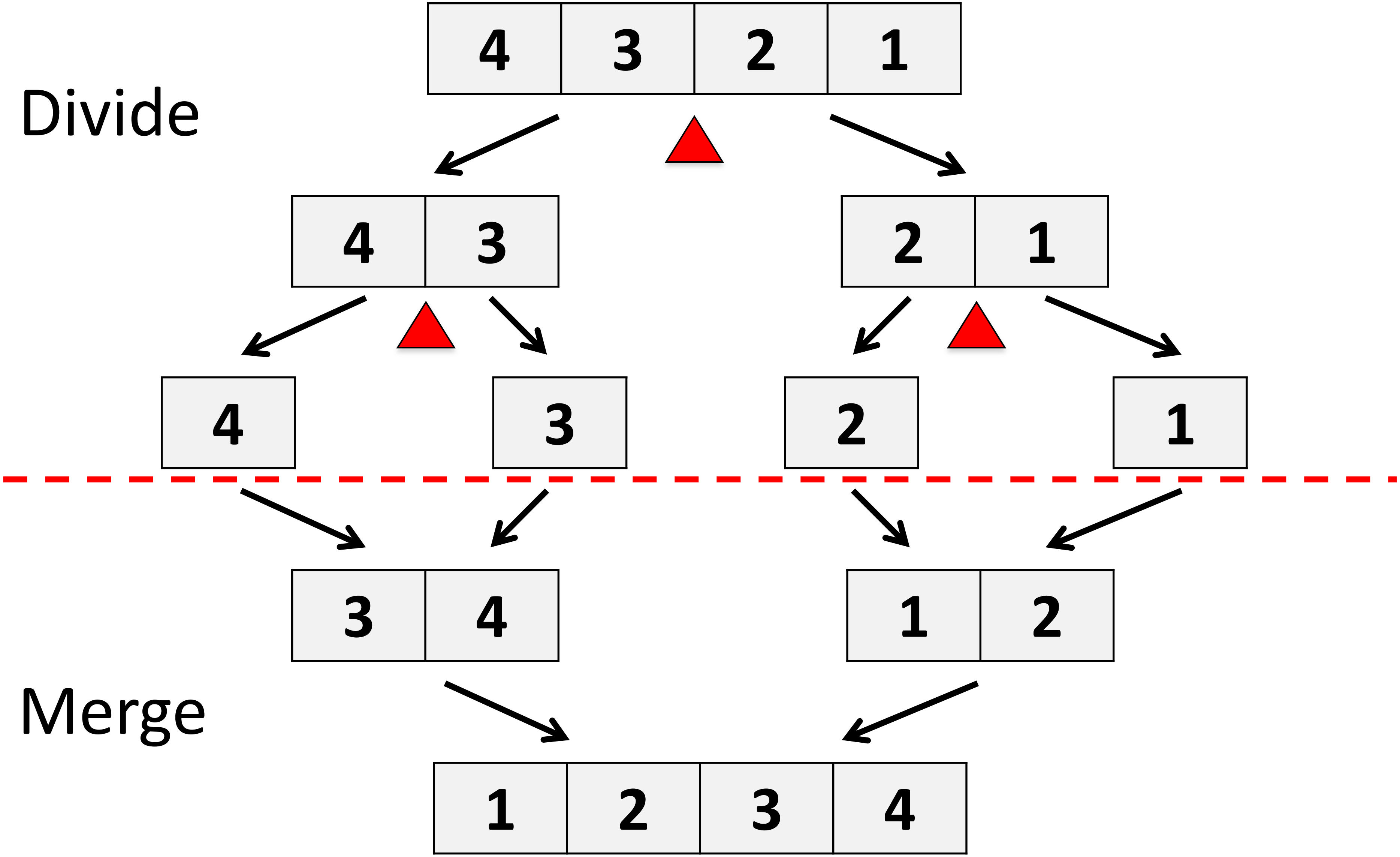
# Merge Sort

**Sort using a divide-and-conquer approach:**

- ***Divide***: Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each.
- ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ***Combine***: Merge the two sorted subsequences to produce the sorted answer.

Note: We will speak more of divide-and-conquer techniques when we will discuss algorithm design.

# Merge Sort (example)


Divide

Merge

# Merge sort (principle)

Recursive case

- Unsorted array A with *n* elements
- Split A in half → 2 arrays L and R with *n/2* elements
- Sort L and R
- Merge the two sorted arrays L and R

Base case: Stop the recursion when the array is of size 1.

Why? Because the array is already sorted!

# Merge-Sort (A, p, r)

**INPUT:** **a sequence of *n* numbers stored in array A**

**OUTPUT:** **an ordered sequence of *n* numbers**

```
MergeSort $(A, p, r)$   // sort $A[p..r]$ by divide & conquer
if $p < r$
    then $q \leftarrow \lfloor (p+r)/2 \rfloor$
        MergeSort $(A, p, q)$
        MergeSort $(A, q+1, r)$
        Merge $(A, p, q, r)$ // merges $A[p..q]$ with $A[q+1..r]$
```

Initial Call: MergeSort($A$, 1, $n$)

# Procedure Merge

```
Merge($A, p, q, r$)
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
    for $i \leftarrow 1$ to $n_1$
        do $L[i] \leftarrow A[p + i - 1]$
    for $j \leftarrow 1$ to $n_2$
        do $R[j] \leftarrow A[q + j]$
    $L[n_1+1] \leftarrow \infty$
    $R[n_2+1] \leftarrow \infty$
    $i \leftarrow 1$
    $j \leftarrow 1$
    for $k \leftarrow p$ to $r$
        do if $L[i] \leq R[j]$
            then $A[k] \leftarrow L[i]$
                $i \leftarrow i + 1$
            else $A[k] \leftarrow R[j]$
                $j \leftarrow j + 1$
```

Copy subarrays n1 and n2 into L and R

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

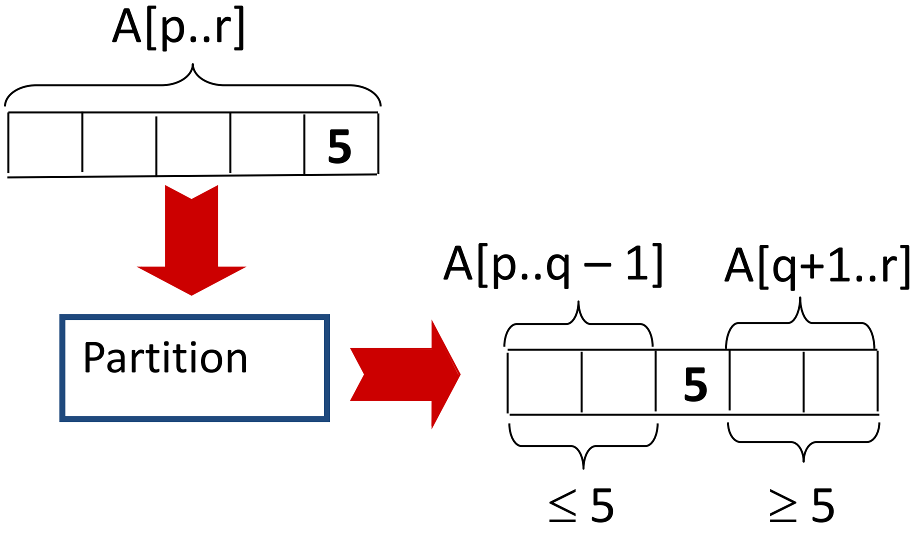
Output: Merged sorted subarray in $A[p..r]$.

**Sentinels**, to avoid having to check if either subarray is fully copied at each step.

# QuickSort

```
Quicksort(A, p, r)
    if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
    fi
```

```
Partition(A, p, r)
    x, i  := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```



Partition stores all the elements lesser than the pivot, then the pivot, then the other elements

# Algorithm analysis

**Q:** How to estimate the running time of a recursive algorithm?

**A:**

1. Define a function T(*n*) representing the time spent by your algorithm to execute an entry of size *n*
2. Write a recursive formula computing T(*n*)
3. Solve the recurrence

Notes:

- *n* can be anything that characterizes accurately the size of the input (e.g., size of the array, number of bits)
- We count the number of elementary operations (e.g., addition, shift) to estimate the running time.
- We usually compute an upper bound rather than exact count.
- We will introduce later a general method to solve this.

# Examples (binary search)

```
int bsearch(int[] A, int i, int j, int x) {
    if (i<=j) { // the region to search is non-empty
        int e = ⌊(i+j)/2⌋;
        if (A[e] > x) { return bsearch(A,i,e-1,x);
        } elif (A[e] < x) { return bsearch(A,e+1,j,x);
        } else { return e; }
    } else { return -1; } // value not found
}
```

$$T(n) = \begin{cases} c & if\ n = 1 \\ T\left(\frac{n}{2}\right) + c' & if\ n > 1 \end{cases}$$

Notes:

- *n* is the size of the array
- Formally, we should use ≤ rather than =

# Example (naïve Fibonacci)

```
public static int Fib(int n) {
    if (n <= 1) { return n; }
    return Fib(n-1) + Fib(n-2);
}
```

$$T(n) = \begin{cases} c & if\ n \leq 1 \\ T(n-1) + T(n-2) + c' & if\ n > 1 \end{cases}$$

What are the value of c and c’ ?

- If $n \leq 1$ there is only one comparison thus *c=1*
- If $n > 1$ there is one comparison and one addition thus *c’=2*

Notes:

- we neglect other constants
- We can approximate *c* and *c’* with an *asymptotic* notation O(1)

# Example (Merge sort)

```
MergeSort (A, p, r)
if (p < r) then
    q ← ⌊(p+r)/2⌋
    MergeSort (A, p, q)
    MergeSort (A, q+1, r)
    Merge (A, p, q, r)
```

- Base case: constant time c
- Divide: computing the middle takes constant time c'
- Conquer: solving 2 subproblems takes $2 \cdot T(n/2)$
- Combine: merging *n* elements takes $k \cdot n$

$$T(n) = \begin{cases} c & if\ n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + k \cdot n + c + c' & if\ n > 1 \end{cases}$$