COMP 251: Recurrences

Jérôme Waldispühl

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
  - 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 credits **+ c(COMP251) + 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(MATH323) = ?

c(MATH323) = 3 credits + c(MATH141)

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

c(MATH141) = ?

c(MATH141) = 4 credits # no prerequisite

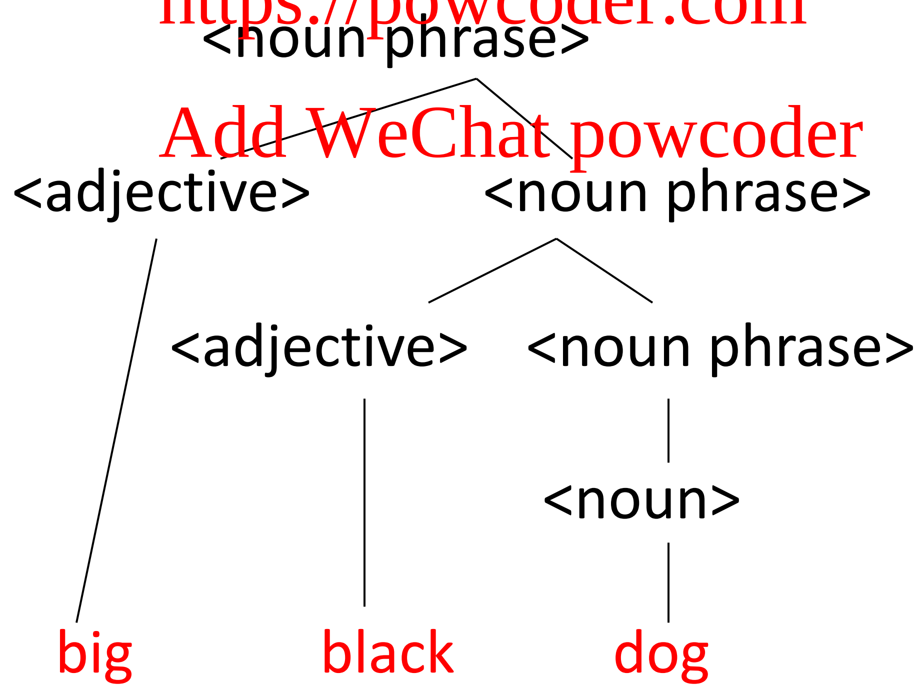
c(COMP462) = 12 credits + 4 credits = 16 credits

Recursive definition

A noun phrase is either

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

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



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)
- Recursion is often a good alternative to iteration (loops).

Definition

Definition (recurrence):

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

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

Examples:

$$T(n) = \begin{cases} 1 & if\ n = 1 \\ T(n-1) + 1 & if\ n > 1 \end{cases}$$

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

Many technical issues:

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

Note: we usually express both the recurrence and its solution using *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

power(7,3) calls

power(7,2) calls

power(7,1) calls

power(7,0) returns **1**

returns 7 * **1** = 7

returns 7 * **7** = 49

returns 7 * **49** = 343

returns 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].

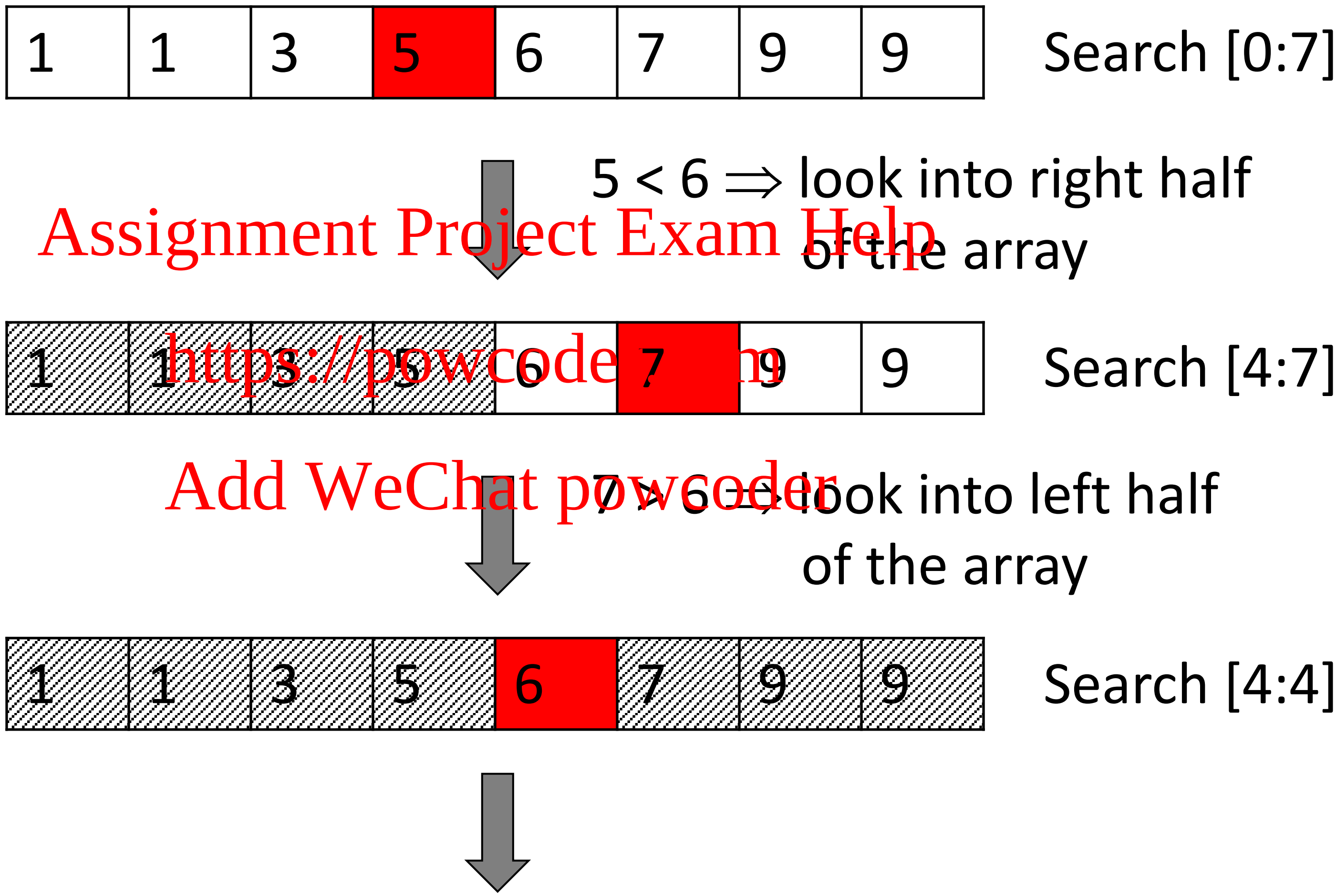
Example: Does the following **sorted** array A contains the number 6?

A =

1	1	3	5	6	7	9	9

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

Binary search example



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)

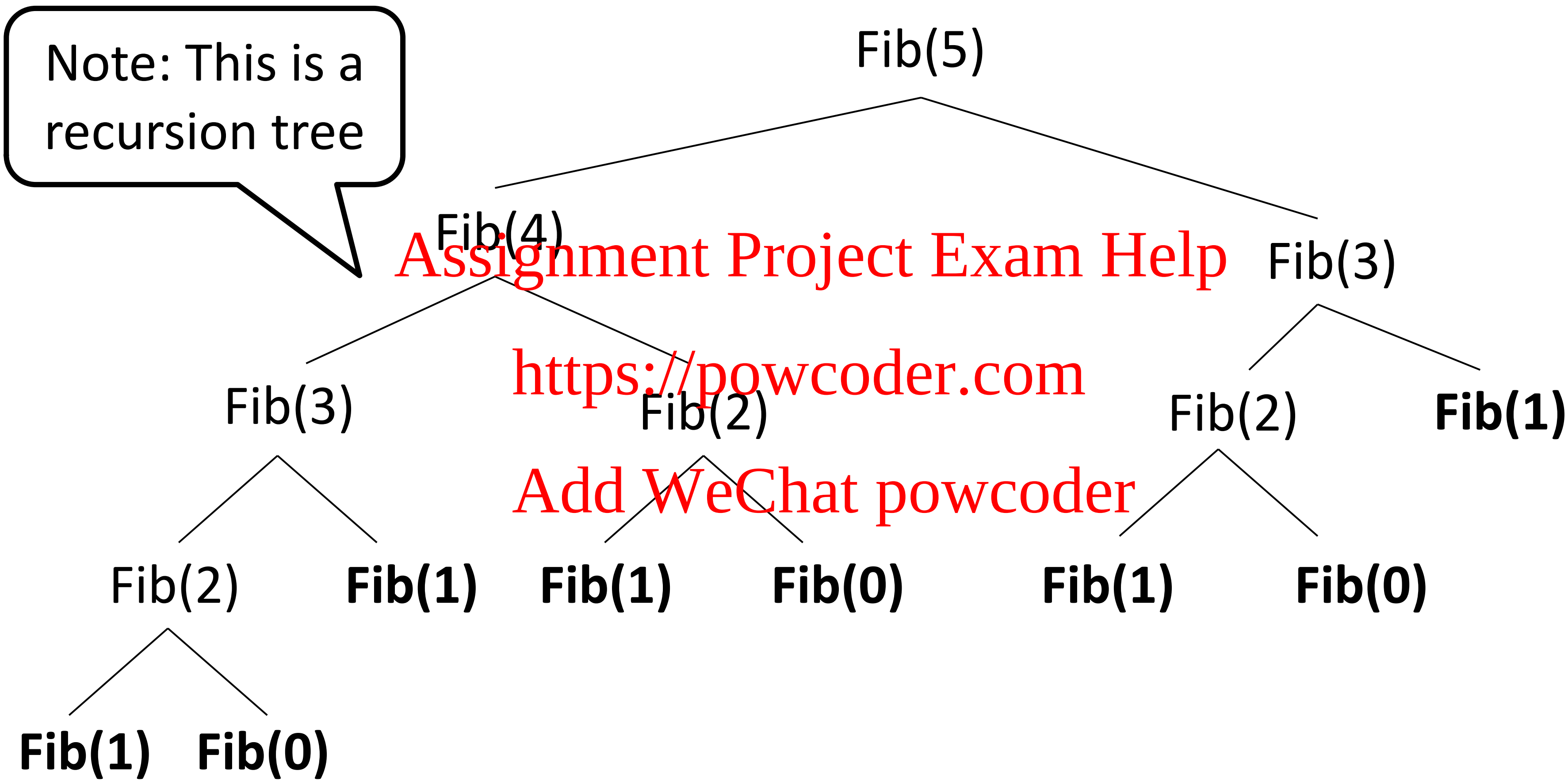
```
public static int Fib(int n) {
   if (n <= 1) {
      return n;
   }
   // {n > 1}
   return Fib(n-1) + Fib(n-2);
}
```

Can handle both base cases together

Recursive case (2 recursive calls)

Note: The algorithm follows almost exactly the definition of Fibonacci numbers.

Recursion is not always efficient!



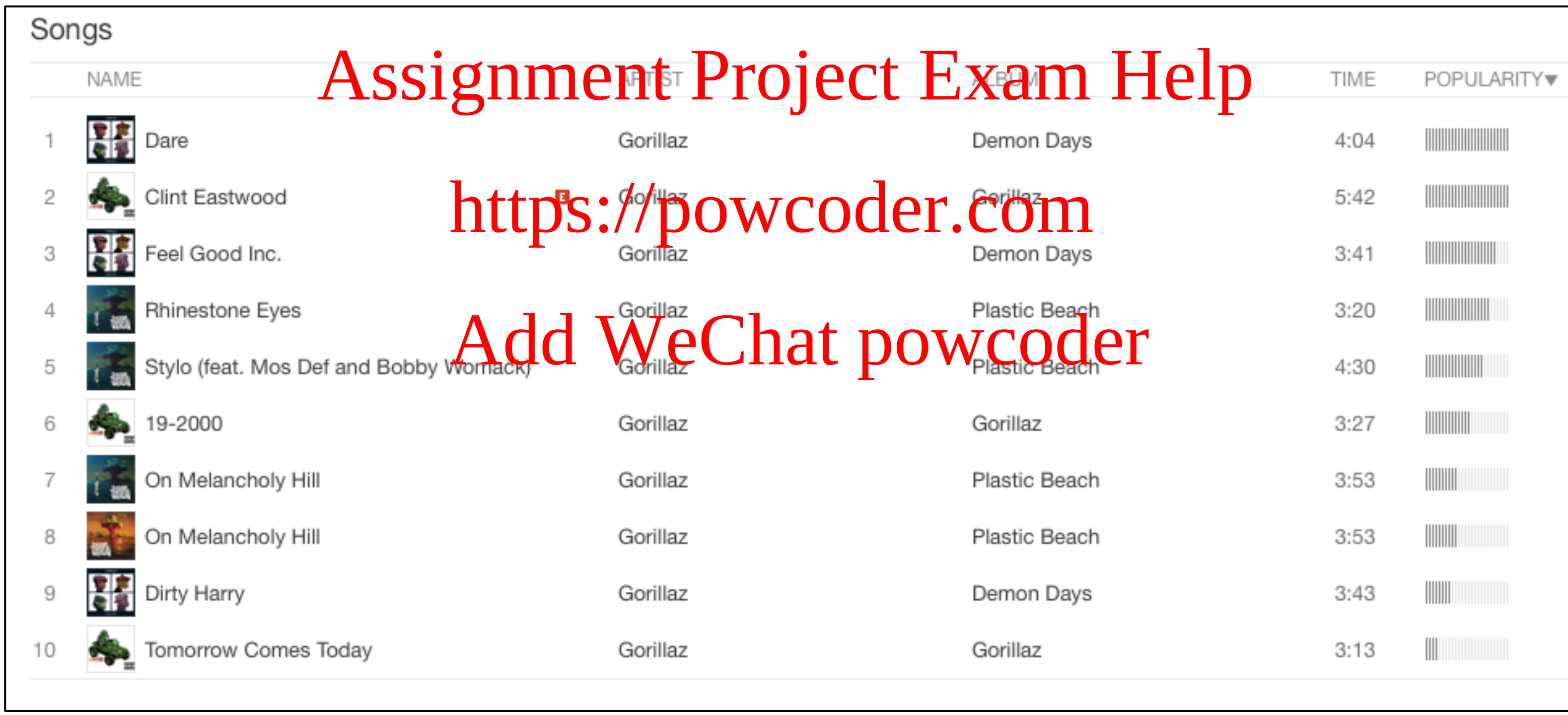
Question: When computing Fib(n), how many times are Fib(0) or Fib(1) called?

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(ϕ^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) = (power(a,n/2))^2 $\Rightarrow$ O(log n) operations

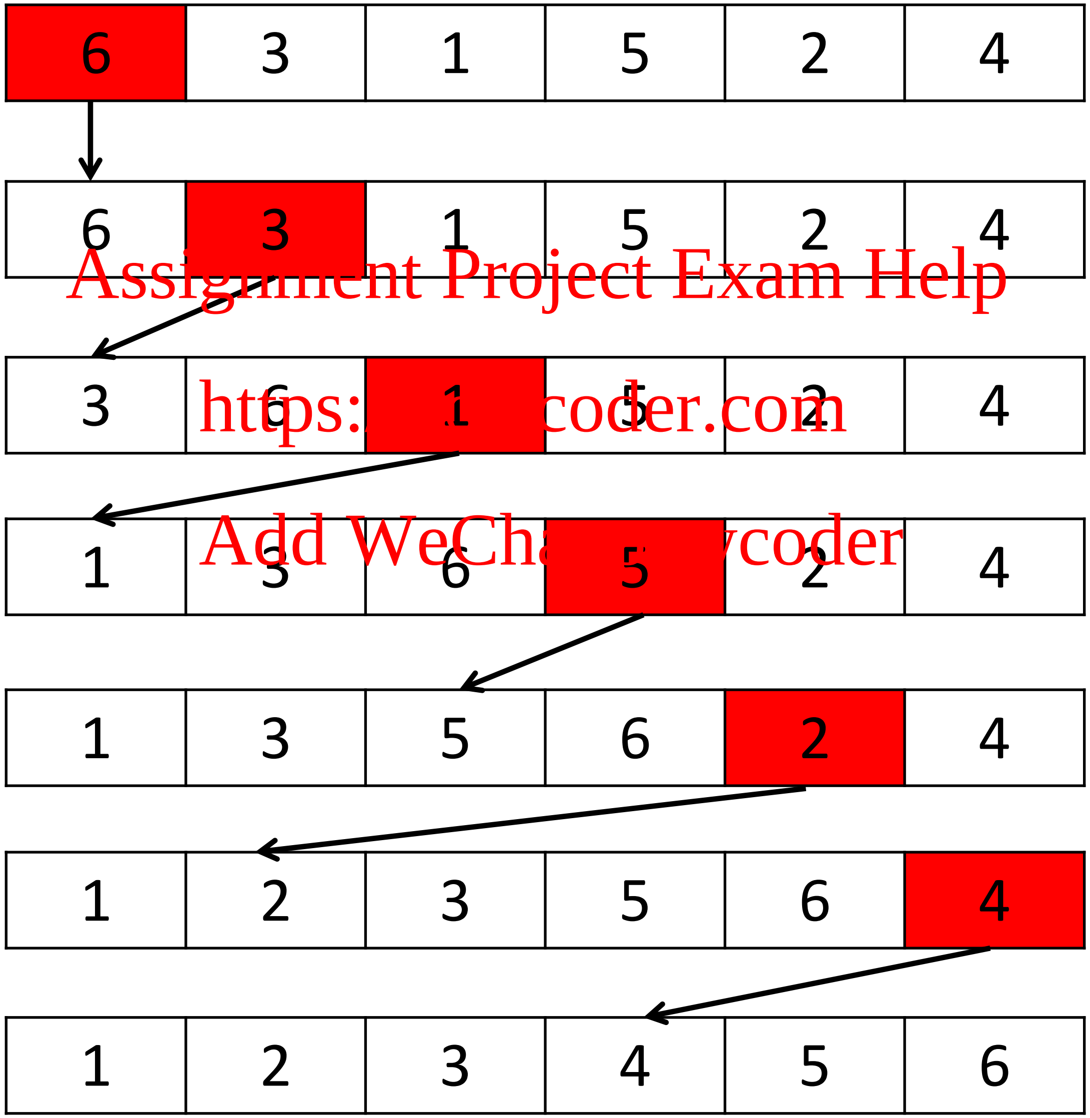
Sorting problem

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

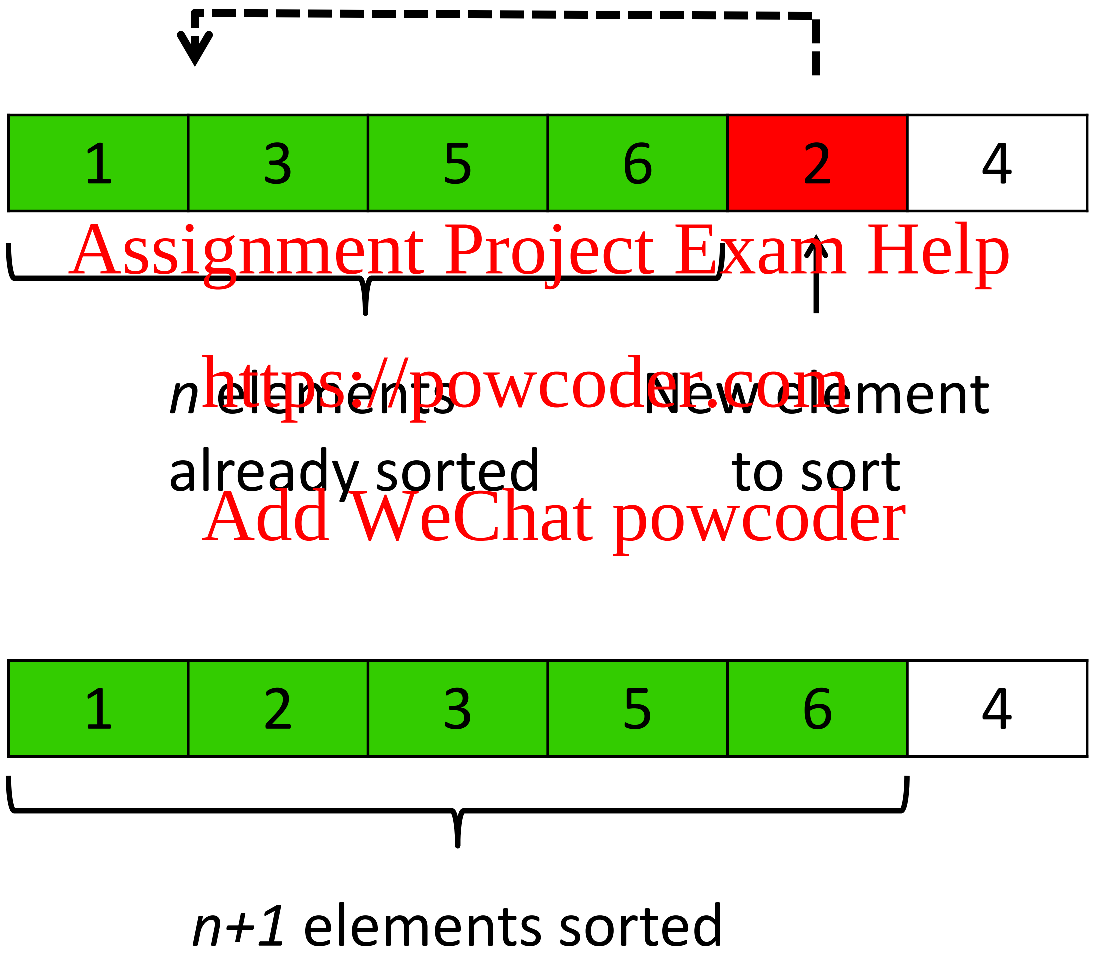


Classical problem in computer science with many different algorithms (bubble sort, merge sort, quick sort, etc.)

Insertion sort



Insertion sort



Insertion sort

```
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!

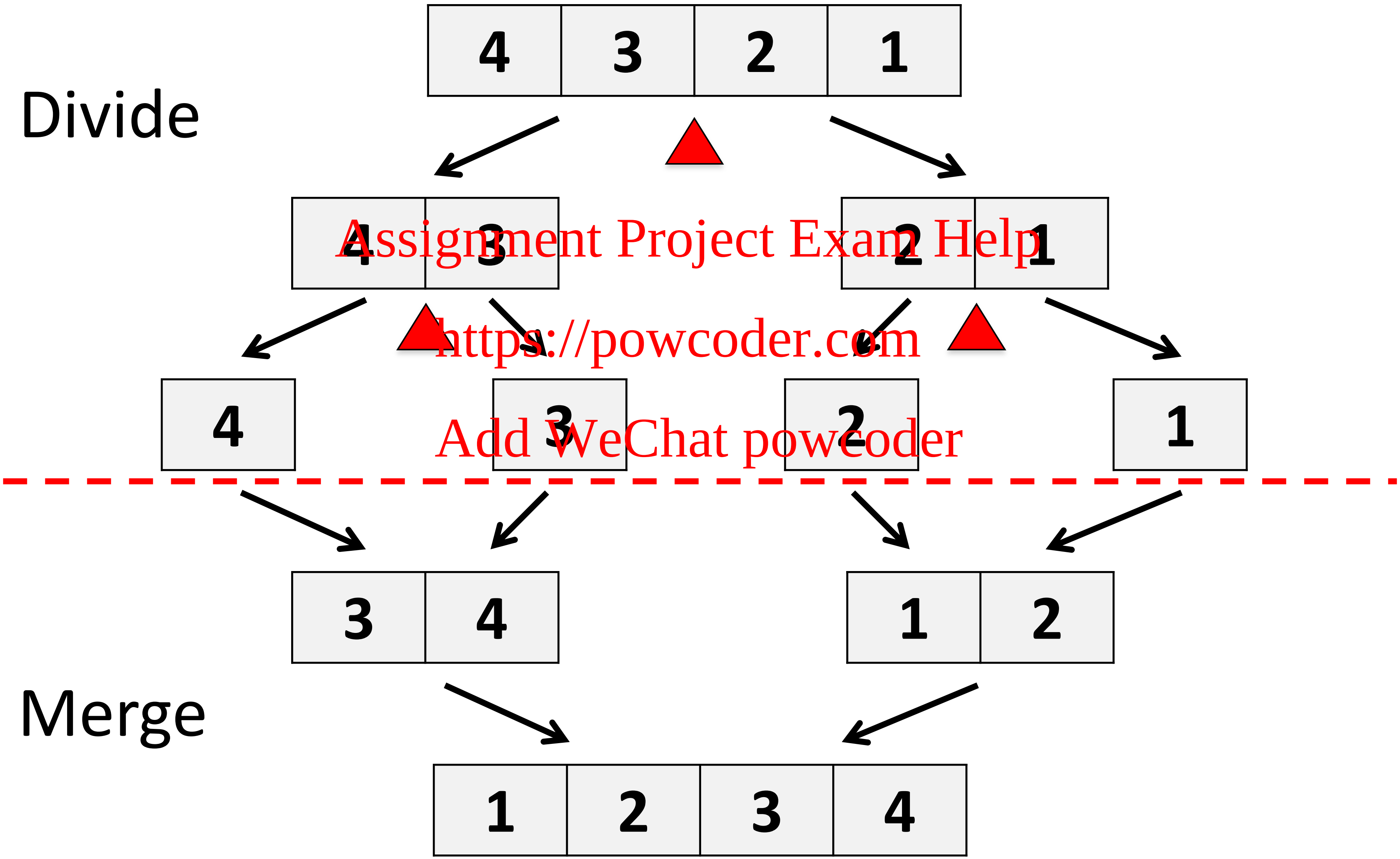
Merge Sort

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.

Merge Sort - Example

Divide



Merge

Merge sort (principle)

Recursive case

- Unsorted array A with *n* elements
- Split A in half $\rightarrow$ 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 ← ⌊(p+r)/2⌋
            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)
n1 ← q – p + 1
n2 ← r – q
    for i ← 1 to n1
        do L[i] ← A[p + i – 1]
    for j ← 1 to n2
        do R[j] ← A[q + j]
    L[n1+1] ← ∞
    R[n2+1] ← ∞
    i ← 1
    j ← 1
    for k ←p to r
        do if L[i] ≤ R[j]
            then A[k] ← L[i]
                    i ← i + 1
            else A[k] ← R[j]
                    j ← j + 1
```

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

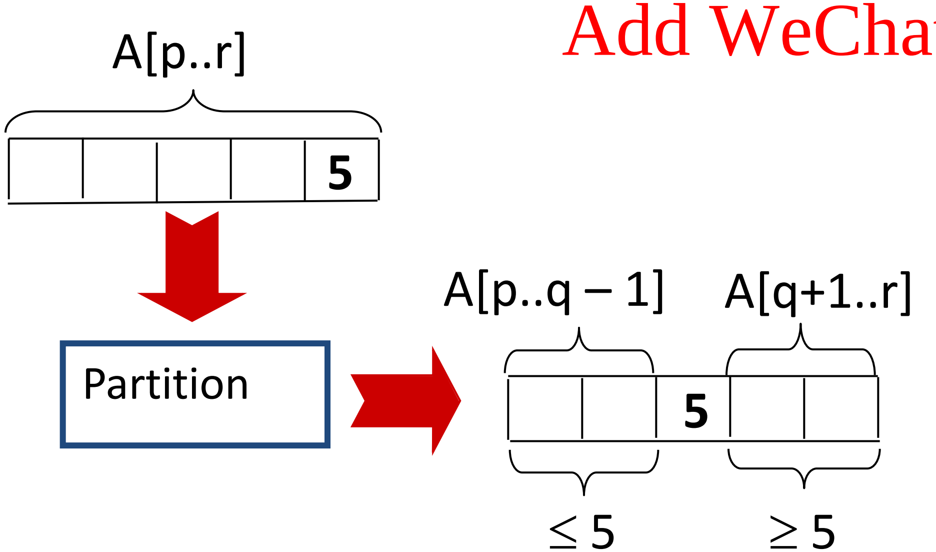
Output: Merged sorted subarray in $A[p..r]$.

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
```

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 often aim to compute an upper bound rather than an exact count.

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 $\leq$ 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 $k \cdot n$

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

Substitution method

How to solve a recursive equation?

1. Guess the solution.
2. Use induction to find the constants and show that the solution works.

Example:

$$T(n) = \begin{cases} 1 & if\ n = 0 \\ 2 \cdot T(n-1) & if\ n > 0 \end{cases}$$

Guess: $T(n) = 2^n$
Base case: $T(0) = 2^0 = 1$ ✓
Inductive case:
Assume $T(n) = 2^n$ until rank *n-1, then show it is true at rank n.*
$T(n) = 2 \cdot T(n-1) = 2 \cdot 2^{n-1} = 2^n$ ✓

Running time of binary search

$$T(n) = \begin{cases} 0 & if\ n = 1 \\ T\left(\frac{n}{2}\right) + 1 & if\ n > 1 \end{cases}$$

Note: set the constant c=0 and c'=1

Guess: $T(n) = \log_2 n$

Base case: $T(1) = \log_2 1 = 0$ ✓

Inductive case:

Assume $T(n/2) = \log_2(n/2)$

Induction hypothesis can be anything < n

$$T(n) = T(n/2) + 1 = \log_2(n/2) + 1$$
$$= \log_2(n) - \log_2 2 + 1 = \log_2 n \checkmark$$

Running time of Merge Sort

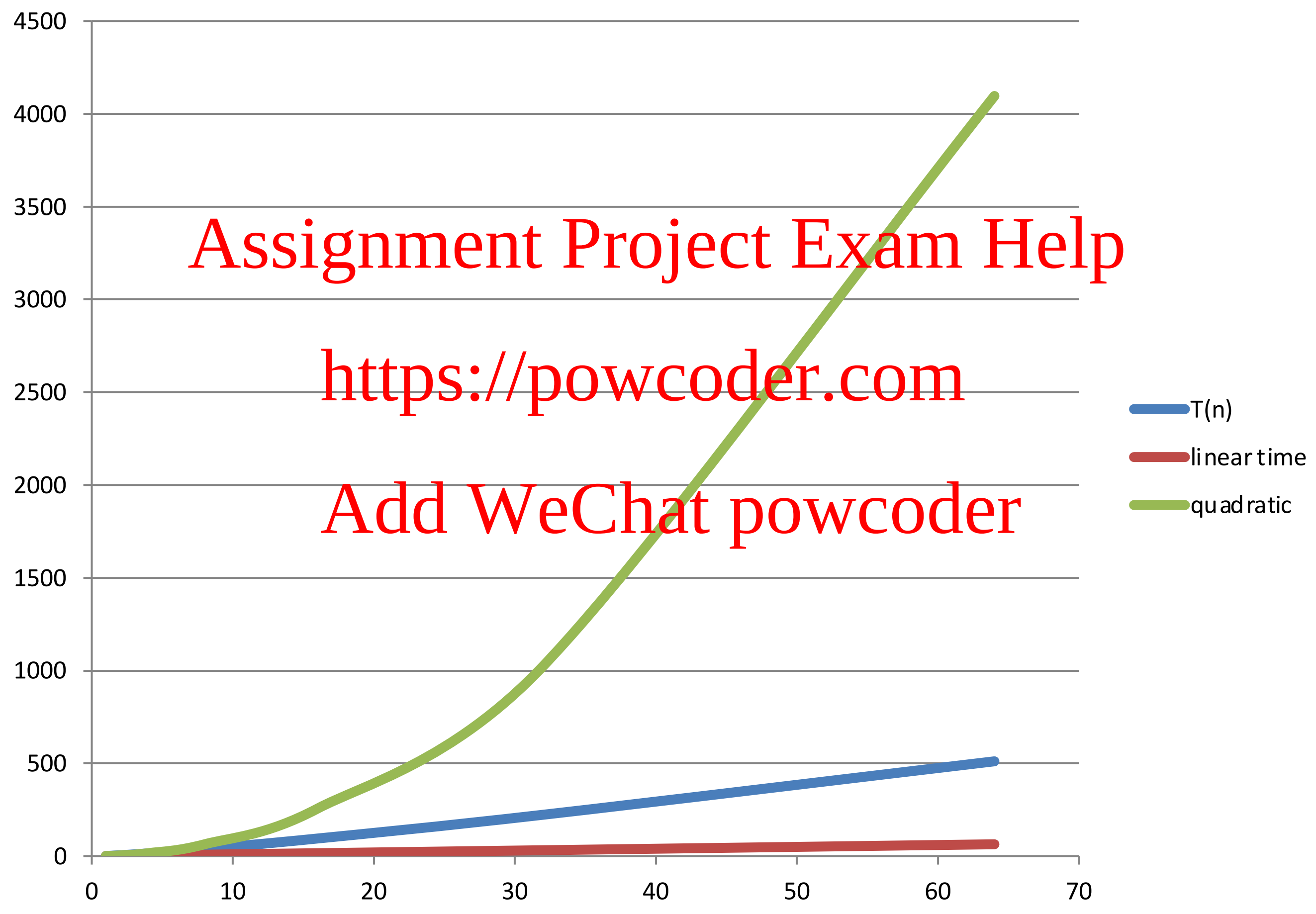
We use a simplified version:

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

Simulation:

n	1	2	4	8	16	32	64	…	n
T(n)	1	5	15	39	95	223	511	…	?

Running time of Merge Sort



Running time of Merge Sort

Guess: $T(n) = \mathrm{n} \cdot \log n + n$

Base case: $T(1) = 1 \cdot \log 1 + 1 = 1$ ✓

Inductive case:

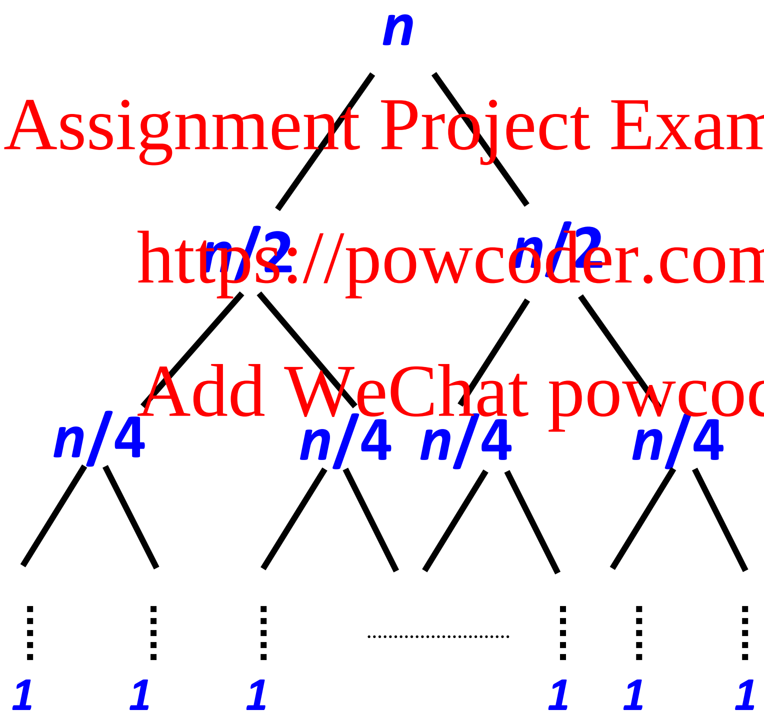
Assume $T(n/2) = \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) + \frac{n}{2}$

$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n = 2 \cdot \left(\frac{n}{2} \cdot \log\left(\frac{n}{2}\right) + \frac{n}{2}\right) + n$
$= n \cdot (\log n - \log 2) + n + n = n \cdot \log n - n + 2 \cdot n$
$= n \cdot \log n + n$ ✓

Note: Here, we use an exact function but it will become simpler when we will use the asymptotic notations

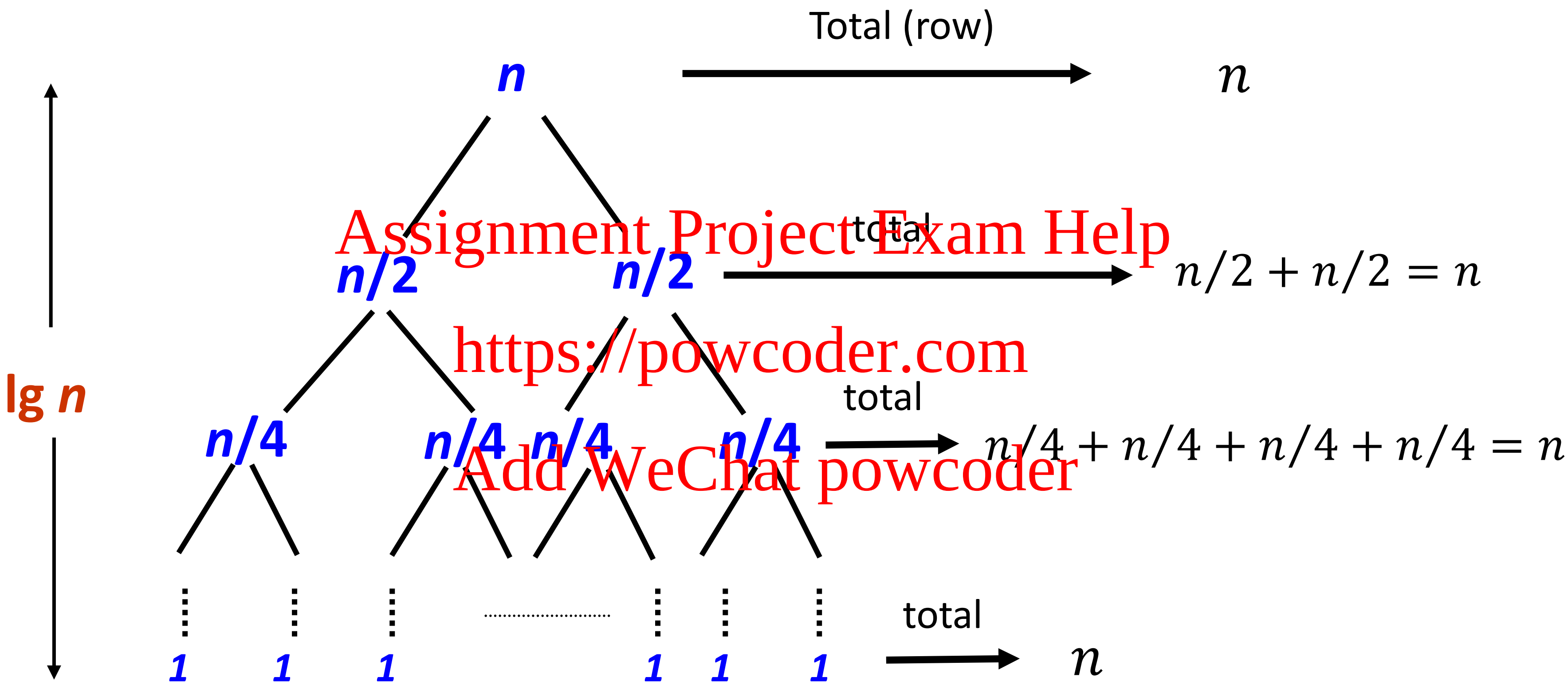
Recursion tree

Objective: Another method to represent the recursive calls and evaluate the running time (i.e. #operations).



- Value of the node is the #operation made by merge
- One branch in the tree per recursive call
- WLOG, we assume that *n* is a power of 2

Recursion tree



Total # operations = total of all rows = $n \times height\ of\ the\ tree$

Q: How many time can we split in half *n*? **A:** $\log n$ *times*