CO ences

Jérôme Wal

School of Computer Science

McGill University

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

Outline

- Introduction: Thinking recursively
- Definition
- E
  - Fibonacci
  - 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 cre **TH323)**

The functi *elf 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

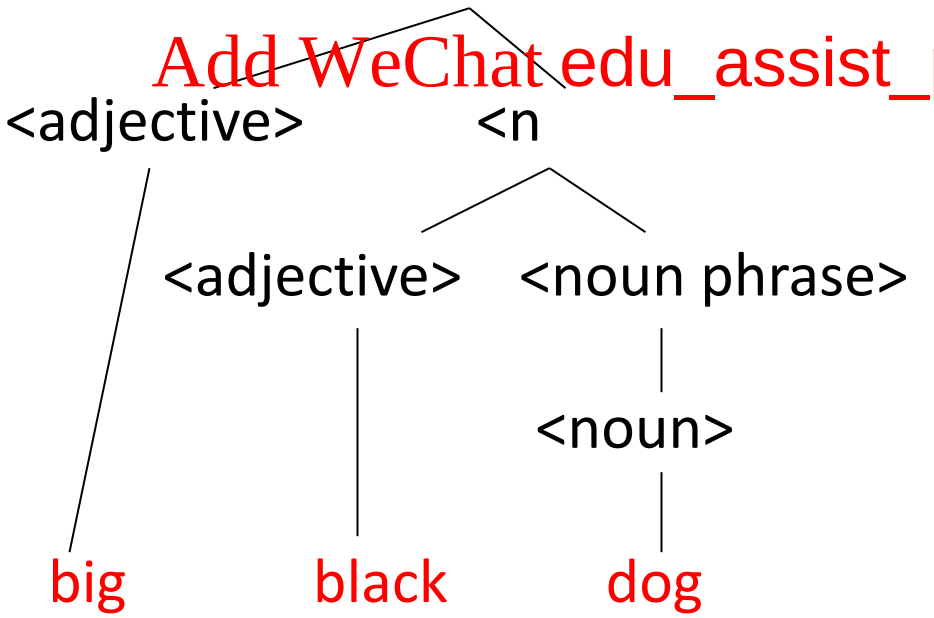
c(COMP462) = 12 credits + 4 credits = 16 credits

Recursive definition

A noun phrase is either

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

<noun phrase> → noun phrase>



Definitions

Recursive definition:

A definition that is defined in terms of itself.

Recursive metho

A method that calls itself (dir irectly).

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 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:

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

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

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:

a, n **Input:** non-

Output: a^n


```
product ← 1
for i = 1 to n do
        product ← product * a
return product
```

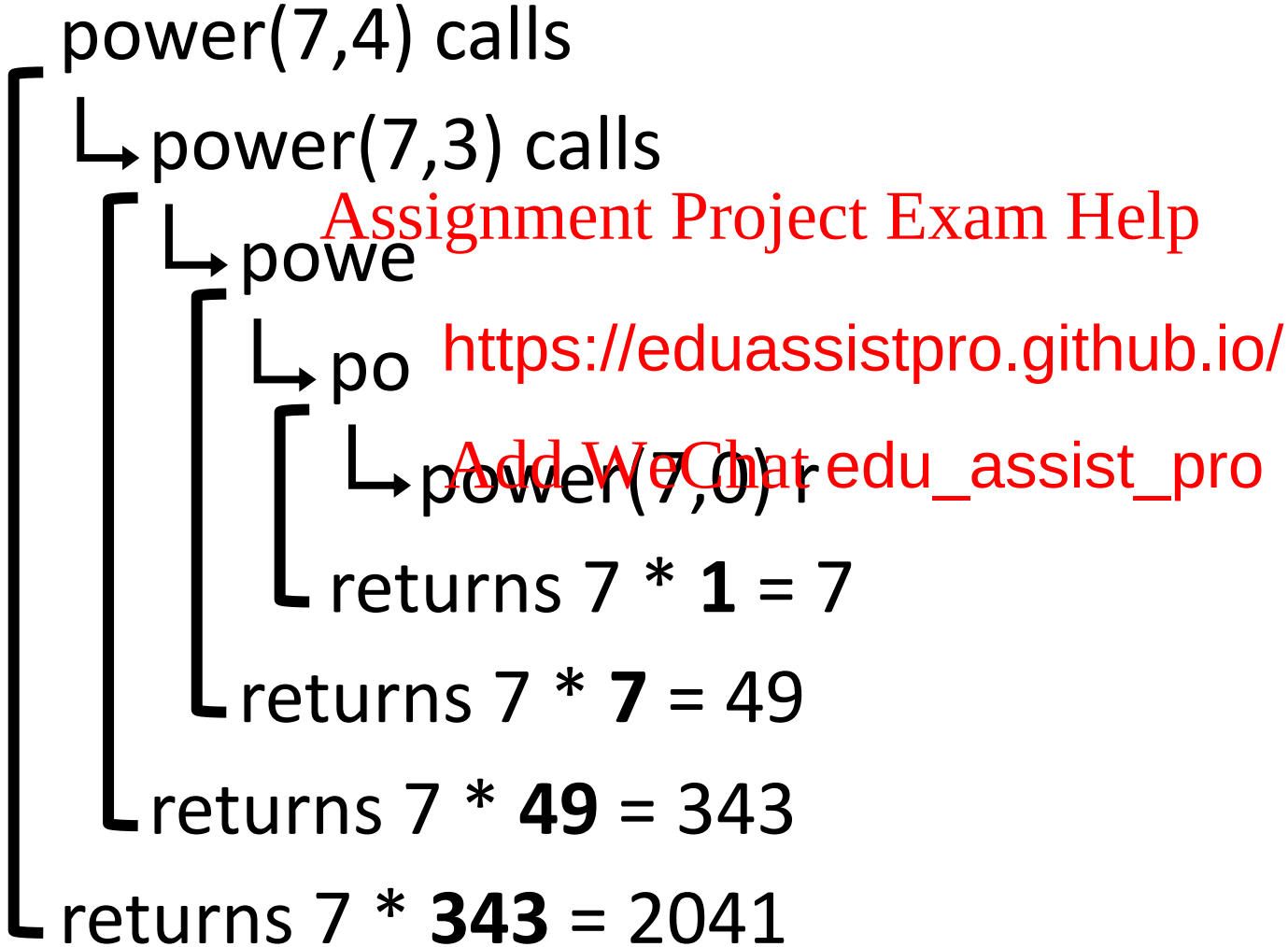
i	0	1	2	3	4
product	1	a	$a * a = a^2$	$a^2 * a = a^3$	$a^3 * a = a^4$

Recursive algorithms

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


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

Example



Algorithm structure

Every recursive algorithm involves at least 2 cases:

base case: A simple occurrence that can be answered directly.

recursive case: ce of the problem that cannot be e described in terms of smaller o f 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

Output: returns th as been found, or returns -1 if the ke p].

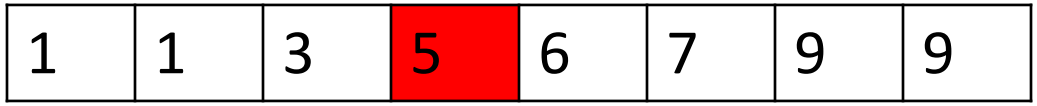
Example: Does the following **sorted** array A contains the number 6?

A =

1	1	3	5	6	7	9	9

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

Binary search example

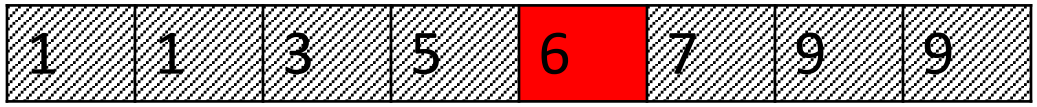


Search [0:7]

$5 < 6 \Rightarrow$ look into right half of the array

Search [4:7]

look into left half of the array



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) {
         retur                    x);
      } else if (A[e] <
         return bsearch(A          );
      } else {
         return e;
      }
   } else { return -1; } // value not found
}
```

Fibonacci numbers

$Fib_0 = 0$ base case

$Fib_1 = 1$ base case

$Fib_n = Fib_{n-1} + Fib$ 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

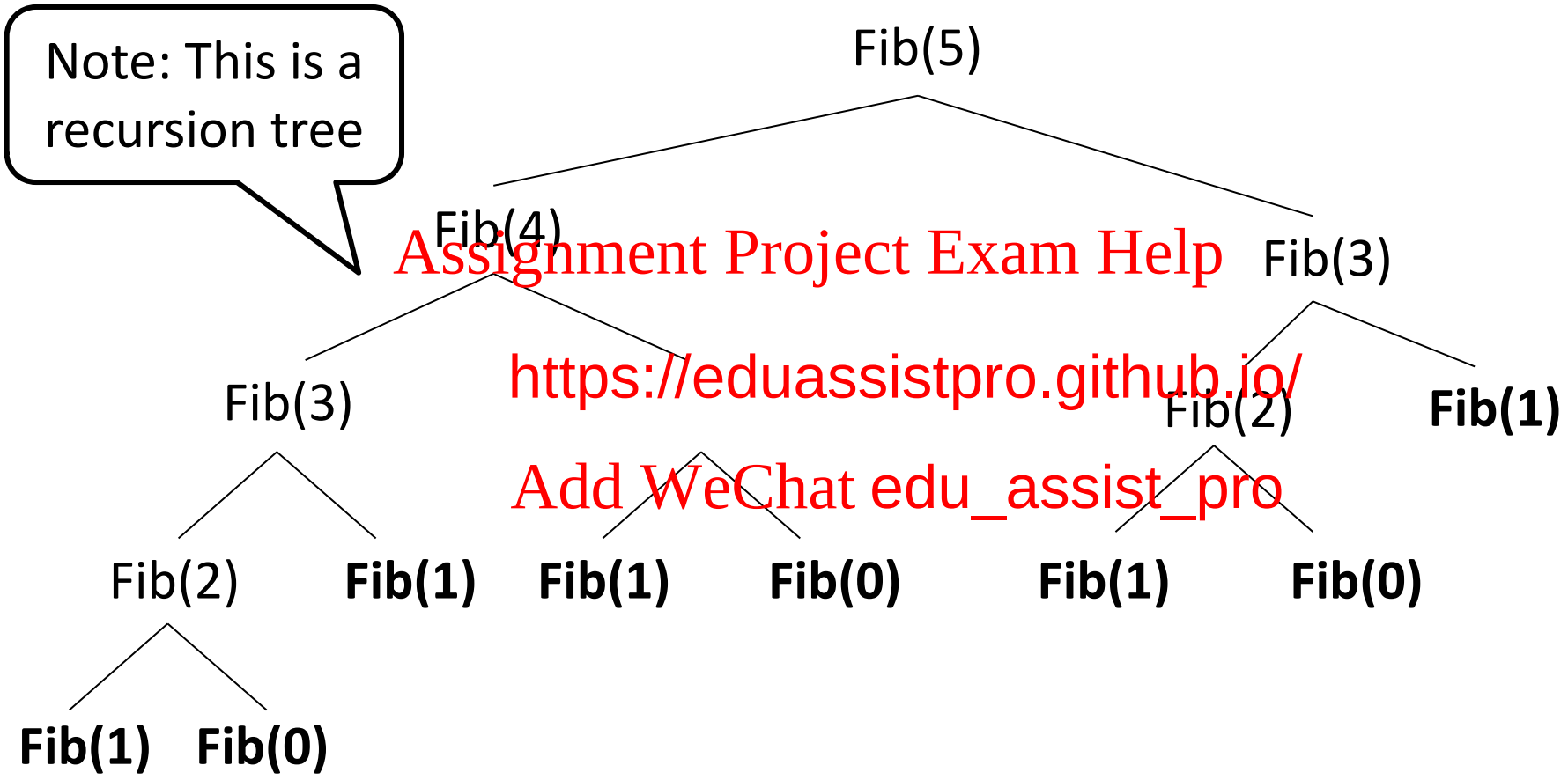
Compute Fibonacci number n (for n ≥ 0)

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


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

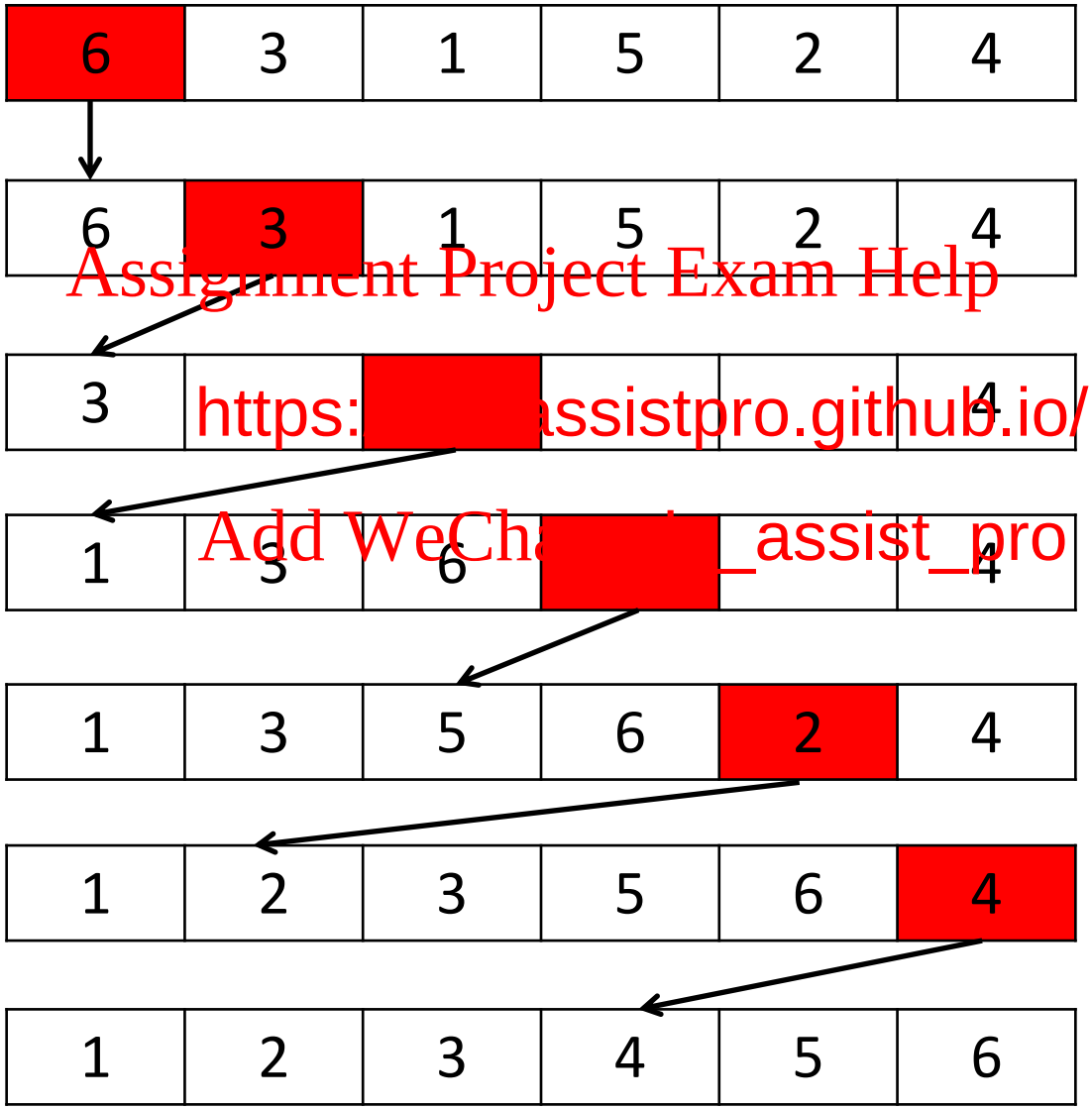
- To write a recursive algorithm:
  - Find how the problem can be broken up in one or more smaller problems of the same nature
  - Remember the
- Naive implem gorithms may lead to prohibitive running time.
  - Naive Fibonacci $\Rightarrow O(\phi^n)$ opera
  - Better Fibonacci $\Rightarrow$ O(log n) operations
- 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

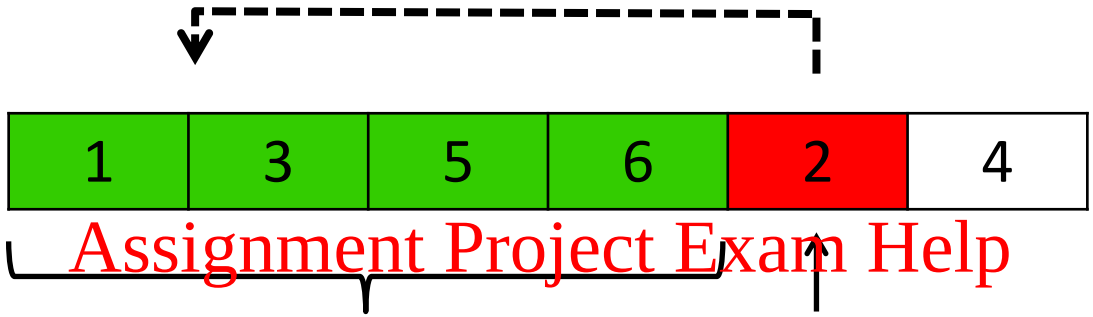
Problem: Given a list of *n* elements from a totally ordered universe, rearrange them in ascending order.

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

Insertion sort

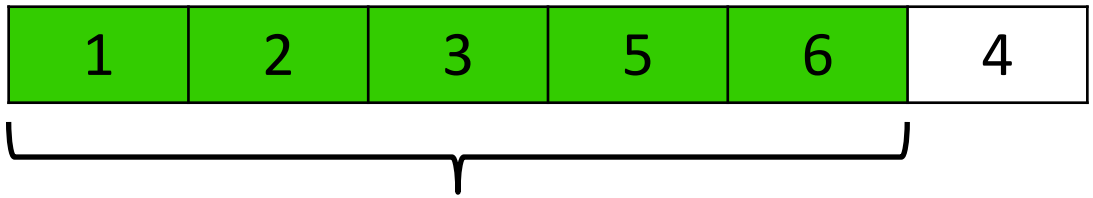


Insertion sort



n elements: *n*+1

already sorted



n+1 elements sorted

Insertion sort

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

    end while
end for
```


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

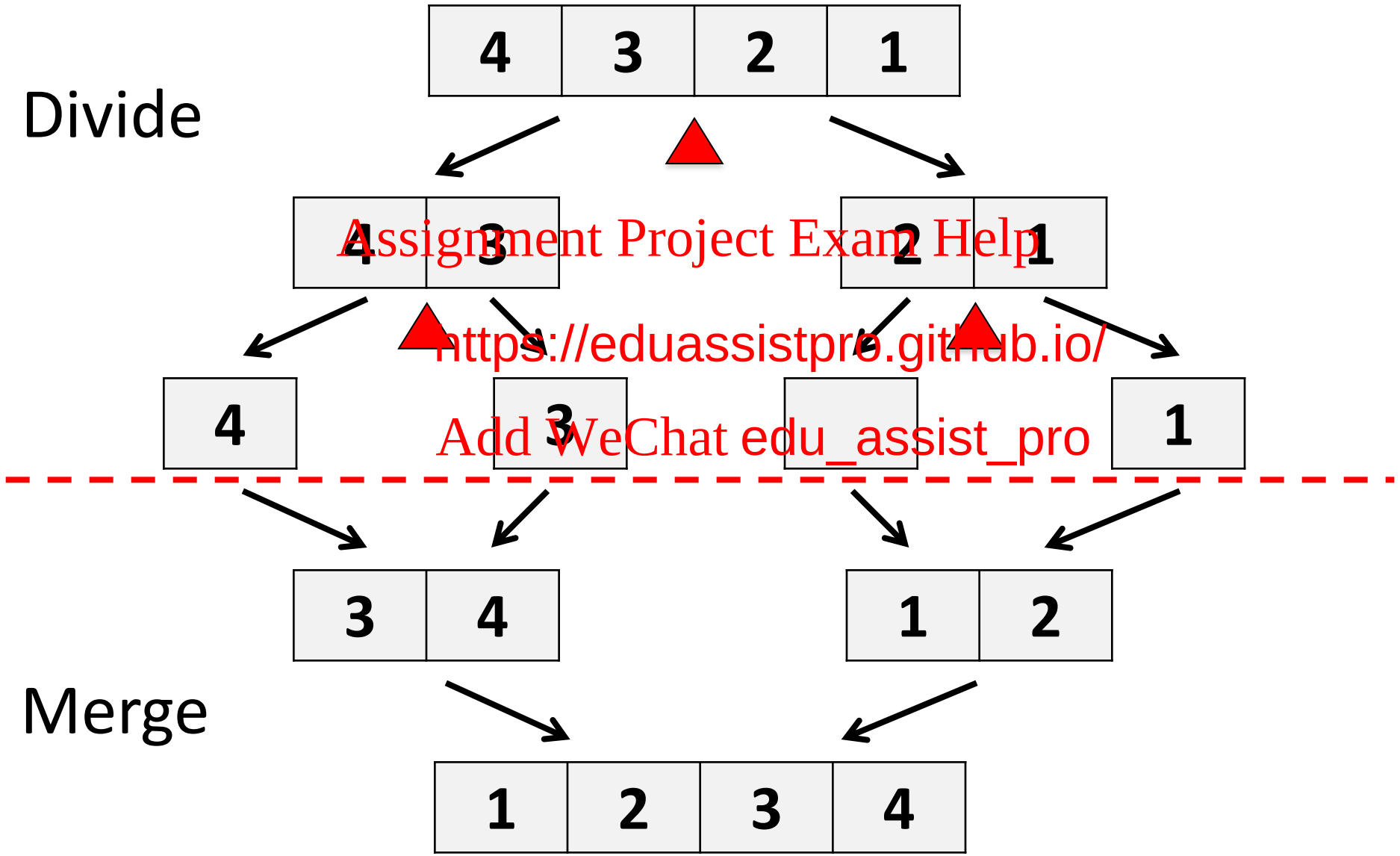
Merge Sort

Sort using a divide-and-conquer approach:

- ***Divide***: Divide the n-element sequence to be sorted into /2 elements each.

- ***Conquer:*** Sort the two su es recursively using merge sort.
- ***Combine***: Merge the two sorted subsequences to produce the sorted answer.

Merge Sort - Example



Merge sort (principle)

Recursive case

- Unsorted array A with *n* elements
- Split A i → with *n/2* elements
- Sort L a
- Merge the two sorted 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
1   if p < r
2       then q ← ⌊(p+r)/2⌋
3            MergeSort (A, p, q)
4            MergeSort (A, q+1, r)
5            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)
1  n1 ← q – p + 1
2  n2 ← r – q
3       for i ← 1 to n1
4          do L[i] ← A[p + i – 1]
5       for j ← 1 to n2
6          do R[j] ← A[q + j]
7       L[n1+1] ← ∞
8       R[n2+1] ← ∞
9       i ← 1
10      j ← 1
11      for k ← p to r
12         do if L[i] ≤ R[j]
13            then A[k] ← L[i]
14                 i ← i + 1
15            else A[k] ← R[j]
16                 j ← j + 1
```

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

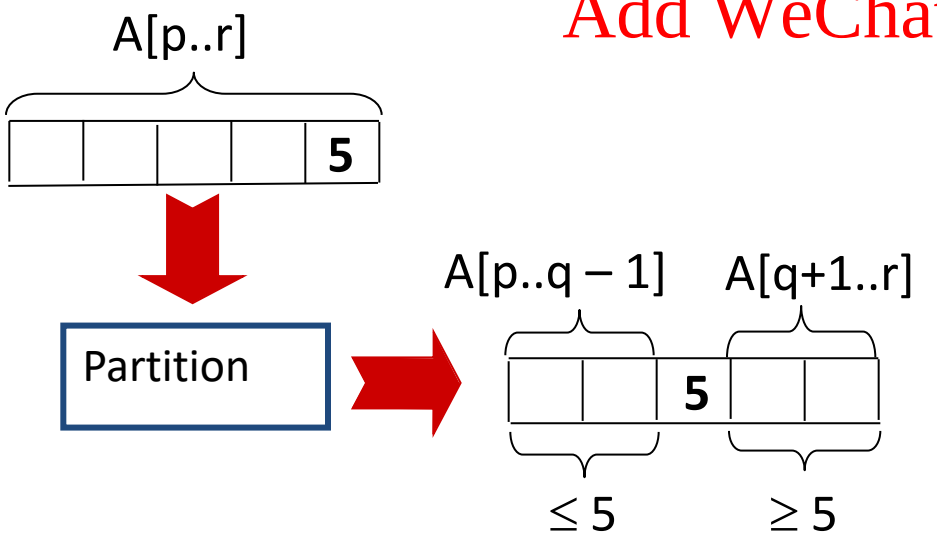
Output: Merged sorted y in $A[p..r]$.

, to avoid having to ither 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);
        Quicksor
    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
    ;
    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 rec T(*n*)
3. Solve the re

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                    arch      (A,e+1,j,x);
        } else {
    } else { return -1; } // va
}
```


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

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);
}
```


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

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
    Merge (A,
```


- Base case: constant time c
- Divide: computing the middle ta t 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:

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

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 con

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

Base case: $T(1) = \log_2 1 = 0$ ✓

Inductive case:

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

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

$= \log_2(n) - \log_2 2 + 1 = \log_2 n$ ✓

duction hypothesis
can be anything < n

Running time of Merge Sort

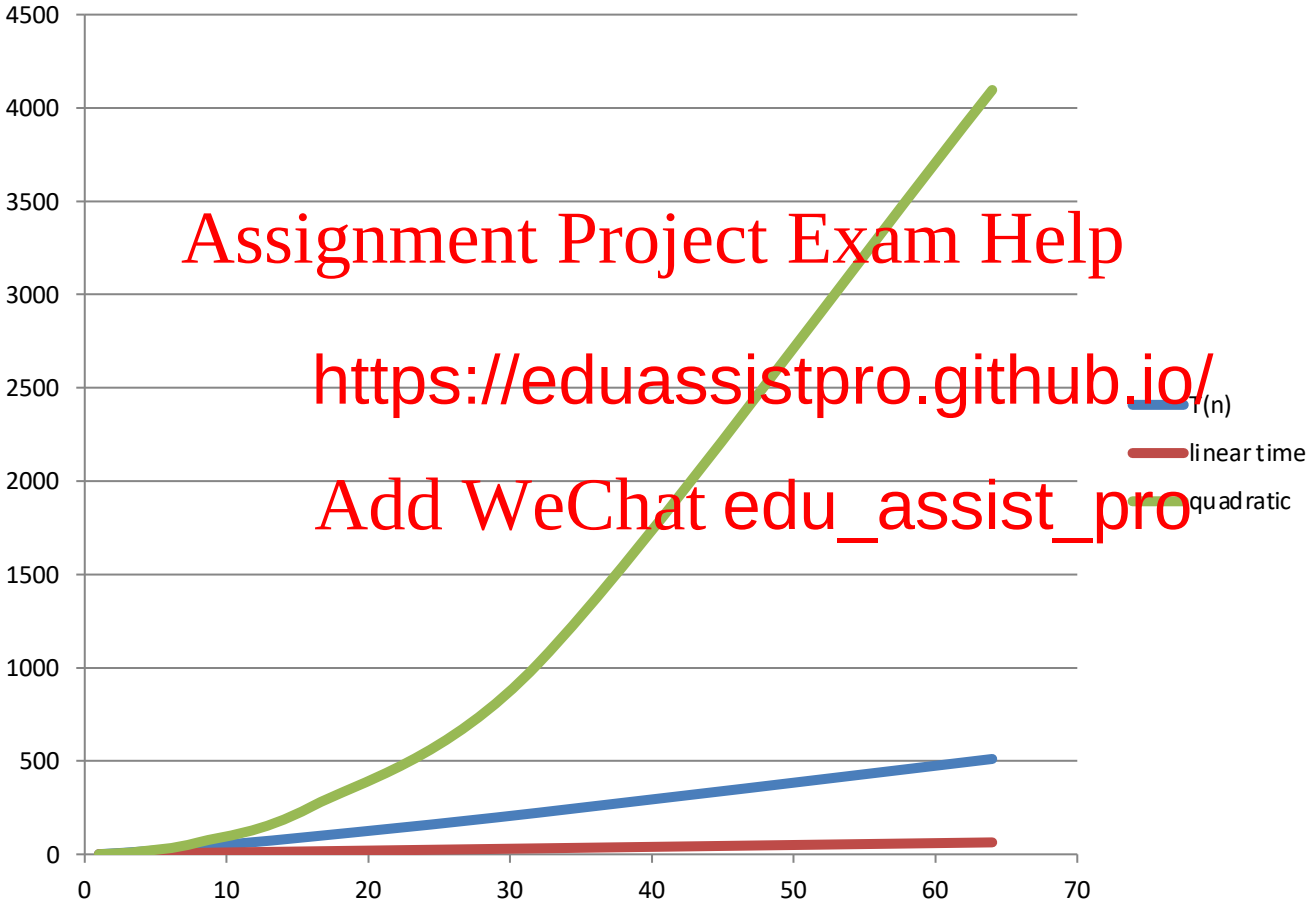
We use a simplified version:

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

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

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

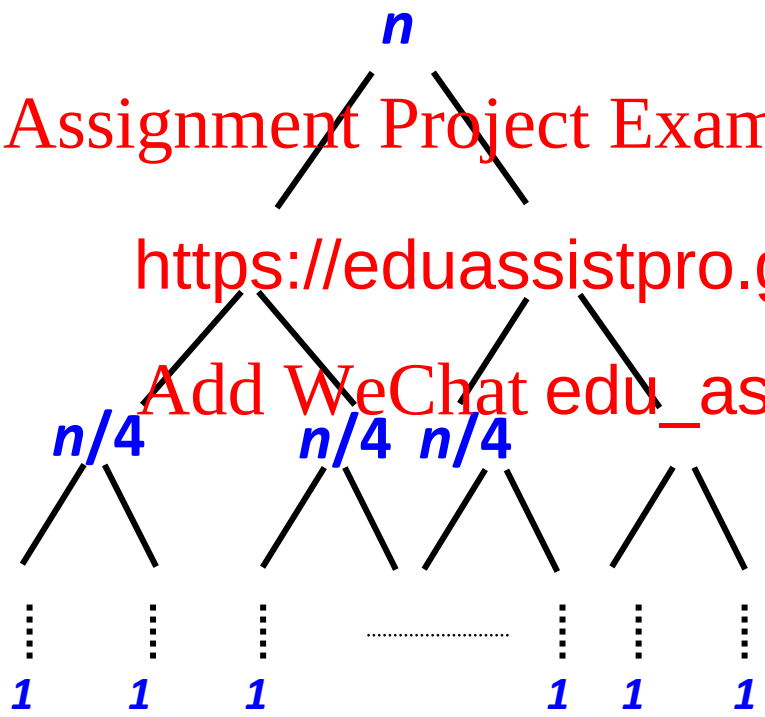
$$= n \cdot (\log n - \log 2) + n + n = n \cdot \log n - n + 2 \cdot n$$

$$= n \cdot \log n + n \checkmark$$

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

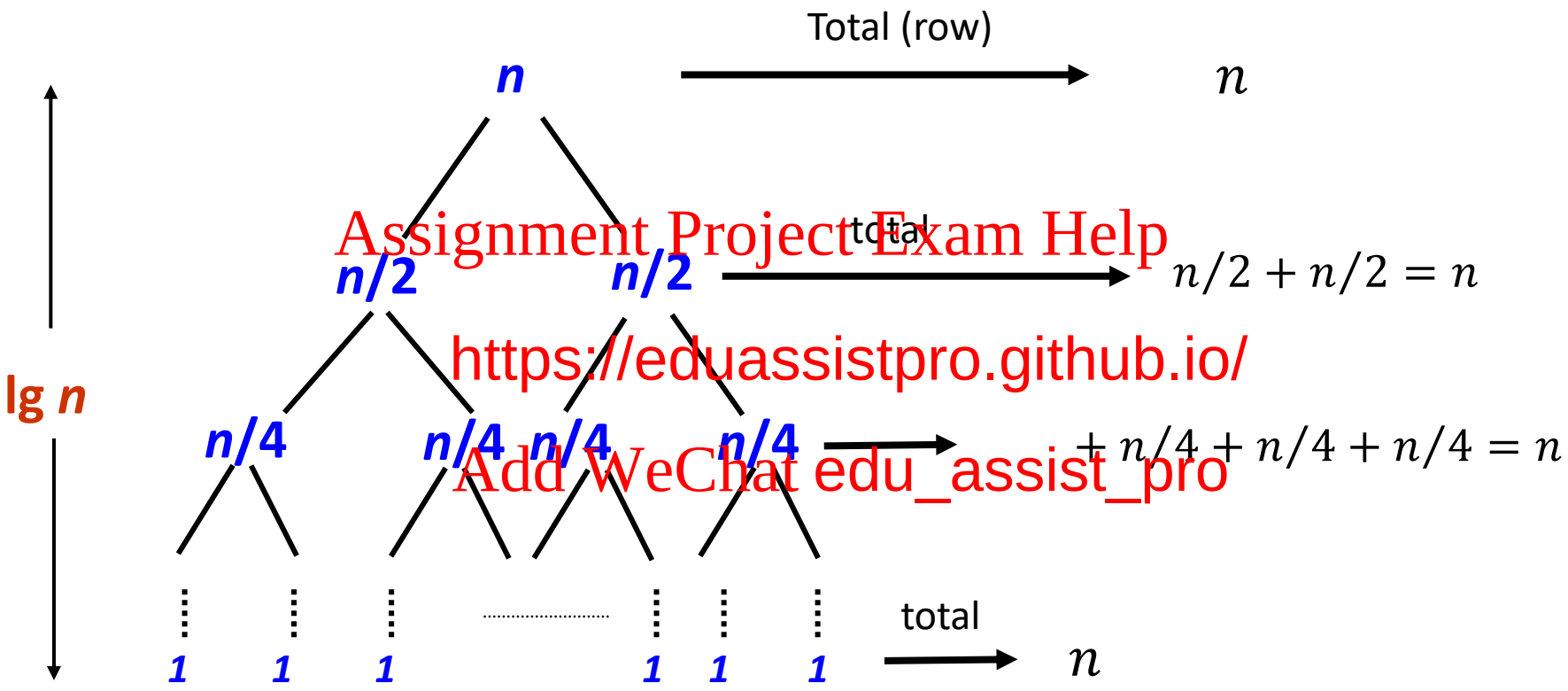
Recursion tree

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*