Complexity of recursive functions

(Weiss 7.5)

Calculating complexity

Let T(n) be the time mergesort takes on a list of size n

Mergesort does O(n) work to split the list in two, two recursive calls of size n/2 and O(n) work to merge the two halves together, so...

$$T(n) = O(n) + 2T(n/2)$$

Time to sort a list of size n

Linear amount of time spent in splitting + merging

Plus two recursive calls of size n/2

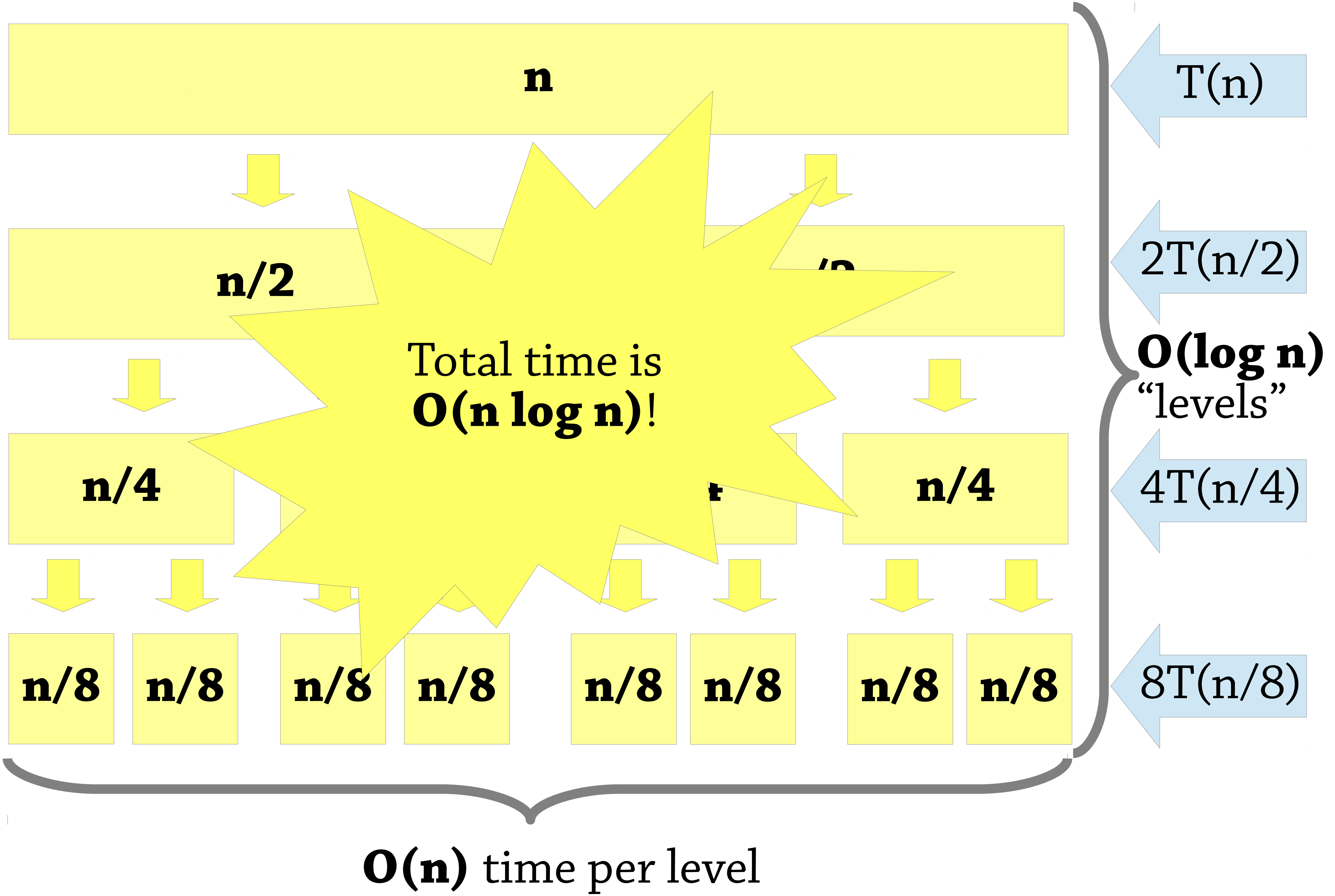
Calculating complexity

Procedure for calculating complexity of a recursive algorithm:

- Write down a *recurrence relation*
  e.g. $T(n) = O(n) + 2T(n/2)$
- *Solve* the recurrence relation to get a formula for $T(n)$ (difficult!)

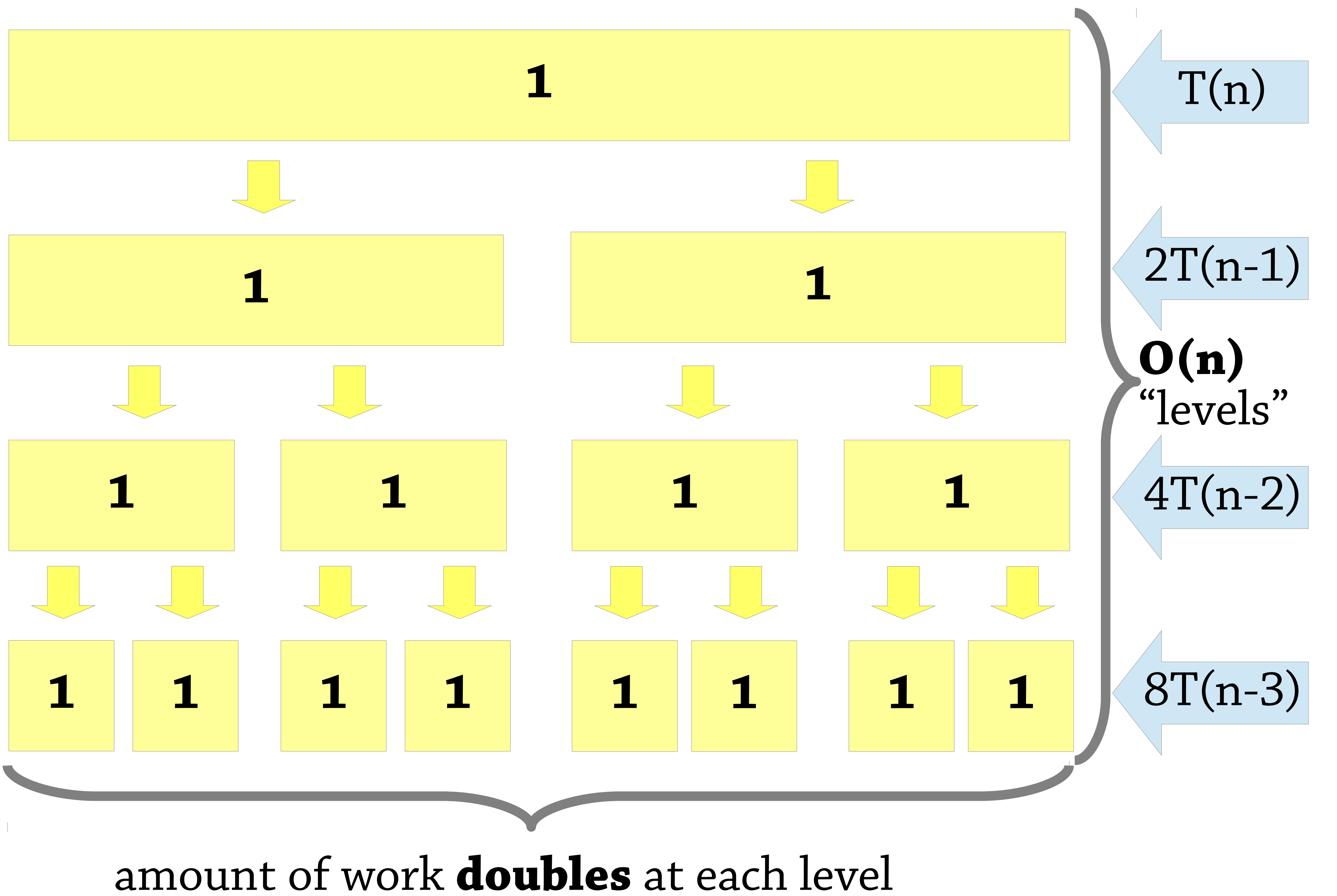
There isn't a general way of solving *any* recurrence relation – we'll just see a few families of them

Approach 1:
draw a diagram

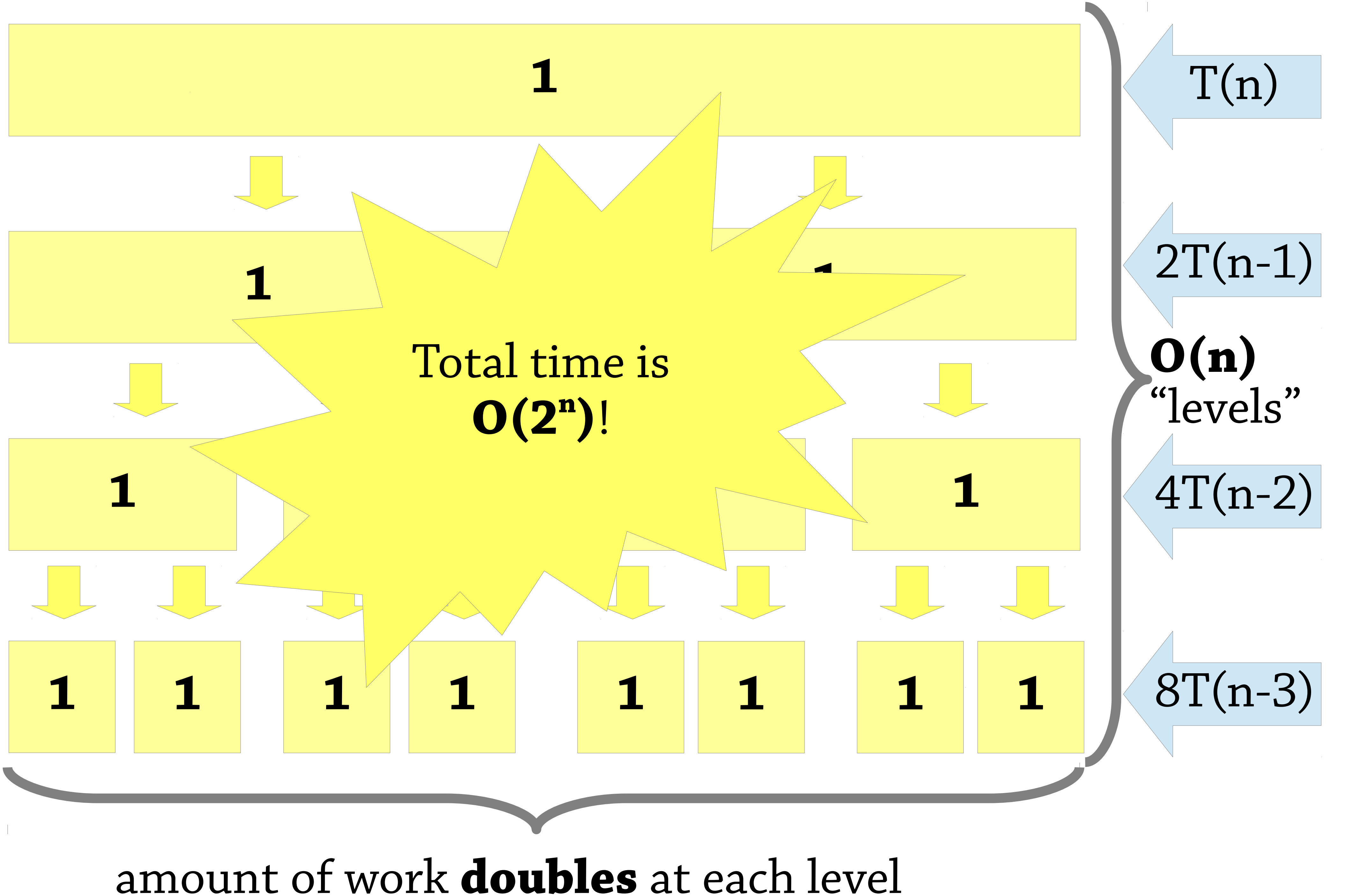

n
n/2
/2
n/4
n/4
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
T(n)
2T(n/2)
4T(n/4)
8T(n/8)
O(log n)
“levels”
O(n) time per level
Total time is
O(n log n)!

Another example:

$$T(n) = O(1) + 2T(n-1)$$



1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
T(n)
2T(n-1)
O(n)
“levels”
4T(n-2)
8T(n-3)
amount of work **doubles** at each level



1
1
1
1
1
1
1
1
1
1
1
1
1
1
Total time is
O(2^n)!
T(n)
2T(n-1)
4T(n-2)
8T(n-3)
O(n)
"levels"
amount of work **doubles** at each level

This approach

Good for building an intuition

Maybe a bit error-prone

Approach 2: *expand out* the definition

Example: solving $T(n) = O(1) + T(n-1)$

Expanding out recurrence relations

$T(n) = 1 + T(n-1)$

$= 2 + T(n-2)$

$= 3 + T(n-3)$

$= \ldots$

$= n + T(0)$

$= O(n)$

T(0) is a constant,
so O(1)

Another example: $T(n) = O(n) + T(n-1)$

$$T(n) = n + T(n-1)$$

$$= n + (n-1) + T(n-2)$$

$$= n + (n-1) + (n-2) + T(n-3)$$

$$= \ldots$$

$$= n + (n-1) + (n-2) + \ldots + 1 + T(0)$$

$$= n(n+1) / 2 + T(0)$$

$$= O(n^2)$$

Another example: $T(n) = O(1) + T(n/2)$

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

$= 2 + T(n/4)$

$= 3 + T(n/8)$

$= \ldots$

$= \log n + T(1)$

$= O(\log n)$

Another example: $T(n) = O(n) + T(n/2)$

$T(n) = n + T(n/2)$:

$T(n) = n + T(n/2)$

$= n + n/2 + T(n/4)$

$= n + n/2 + n/4 + T(n/8)$

$= \ldots$

$= n + n/2 + n/4 + \ldots$

$< 2n$

$= O(n)$

Functions that recurse once

$T(n) = O(1) + T(n-1)$: $T(n) = O(n)$

$T(n) = O(n) + T(n-1)$: $T(n) = O(n^2)$

$T(n) = O(1) + T(n/2)$: $T(n) = O(\log n)$

$T(n) = O(n) + T(n/2)$: $T(n) = O(n)$

An *almost-rule-of-thumb*:

- Solution is *maximum recursion depth* times *amount of work in one call*

(except that this rule of thumb would give $O(n \log n)$ for the last case)

Divide-and-conquer algorithms

$T(n) = O(n) + 2T(n/2)$: $T(n) = O(n \log n)$

- This is mergesort! There is a nice proof in the book (theorem 7.4).

$T(n) = 2T(n-1)$: $T(n) = O(2^n)$

- Because 2^n recursive calls of depth n

Other cases: *master theorem* (Wikipedia) or theorem 7.5 from book

- Kind of fiddly – best to just look it up if you need it

Complexity of recursive functions

Basic idea – recurrence relations

Easy enough to write down, hard to solve

- One technique: expand out the recurrence and see what happens
- Another rule of thumb: multiply work done per level with number of levels
- Drawing a diagram (like for quicksort) can help!

Master theorem for divide and conquer

Luckily, in practice you come across the same few recurrence relations, so you just need to know how to solve those