CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Data structures

Complexity - continued

Dr. Alex Gerdes

DIT961 - VT 2018

The story so far

Big-O notation: drops constant factors in algorithm runtime

- $O(n^2)$: time proportional to square of input size (e.g. ???)
- $O(n)$: time proportional to input size (e.g. ???)
- $O(\log n)$: time proportional to log of input size, or: time proportional to n, for input of size 2^n (e.g. ???)

We also accept answers that are too big so something that is $O(n)$ is also $O(n^2)$

The story so far

Big-O notation: drops constant factors in algorithm runtime

- $O(n^2)$: time proportional to square of input size (e.g. naïve dynamic arrays)
- $O(n)$: time proportional to input size (e.g. linear search, good dynamic arrays)
- $O(\log n)$: time proportional to log of input size, or: time proportional to n, for input of size 2^n (e.g. binary search)

We also accept answers that are too big so something that is $O(n)$ is also $O(n^2)$

The story so far

Hierarchy

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

- Adding together terms gives you the *biggest* one

- e.g., $O(n) + O(\log n) + O(n^2) = O(n^2)$

Computing big-O using hierarchy:

- $2n^2 + 3n + 2 = ???$

The story so far

Hierarchy

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

- Adding together terms gives you the *biggest* one

- **e.g.,** $O(n) + O(\log n) + O(n^2) = O(n^2)$

Computing big-O using hierarchy:

- $2n^2 + 3n + 2 = O(n^2) + O(n) + O(1) = O(n^2)$

Multiplying big-O

$O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$

- e.g., $O(n^2) \times O(\log n) = O(n^2 \log n)$

You can drop constant factors:

- $k \times O(f(n)) = O(f(n))$, if k is constant e.g. $2 \times O(n) = O(n)$

(Exercise: show that these are true)

The rules

There are three rules you need for calculating big-O:

- Addition (hierarchy)
- Multiplication
- Replacing a term with a *bigger* term

Quiz

What is $(n^2 + 3)(2^n \times n) + \log_{10} n$ in big-O notation?

Answer

$$(n^2 + 3)(2^n \times n) + \log_{10} n$$

$$= \mathrm{O}(n^2) \times \mathrm{O}(2^n \times n) + \mathrm{O}(\log n)$$

$$= \mathrm{O}(2^n \times n^3) + \mathrm{O}(\log n) \qquad \{\mathit{multiplication}\}$$

$$= \mathrm{O}(2^n \times n^3) \qquad \{\mathit{hierarchy}\}$$

Example of replacing a term

Suppose we want to prove from scratch the rules for adding big-O:

- $O(n^2) + O(n^3) = O(n^3)$

We know $n^2 < n^3$

$O(n^2) + O(n^3)$

$\rightarrow O(n^3) + O(n^3)$ {*since* $n^2 < n^3$}

$= 2 \times O(n^3)$

$= O(n^3)$ {*throw out constant factors*}

Complexity of a program

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < a.length; j++)
      if (a[i].equals(a[j]) && i != j)
        return false;
  return true;
}
```

Outer loop runs n times:
$O(n) \times O(n) = O(n^2)$

Inner loop runs n times:
$O(n) \times O(1) = O(n)$

Loop body:
$O(1)$

Complexity of loops

The complexity of a loop is:

- the number of times it runs
  times the complexity of the body

Or:

- If a loop runs $O(f(n))$ times
  and the body takes $O(g(n))$ time
  then the loop takes $O(f(n) \times g(n))$

What about this one?

Outer loop runs n^2 times:
$\mathbf{O(n^2)} \times O(n) = O(n^3)$

```
void function(int n) {
  for (int i = 0; i < n*n; i++)
    for (int j = 0; j < n; j++)
      "something taking O(1) time";
}
```

Inner loop runs n times:
$O(n) \times O(1) = O(n)$

Loop body:
O(1)

What about this one?

Outer loop runs n^2 times:
$O(n^2) \times O(n) = O(n^3)$

```
void function(int n) {
  for (int i = 0; i < n*n; i++)
    for (int j = 0; j < n/2; j++)
      ”something taking O(1) time”;
}
```

Inner loop runs $\boldsymbol{n/2}$ times:
$O(n) \times O(1) = \mathbf{O(\boldsymbol{n})}$

Loop body:
$O(1)$

Here's a new one

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < i; j++)
      if (a[i].equals(a[j]))
        return false;
  return true;
}
```

Inner loop is
$i \times O(1) = O(i)$???
But it should be in terms of n?

Loop body:
$O(1)$

Here's a new one

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 1; j < i; j++)
      if (a[i].equals(a[j]))
        return false;
  return true;
}
```

Outer loop runs n times:
$O(n) \times O(n) = O(n^2)$

$i < \mathrm{n}$, so ***i* is $\mathbf{O(n)}$**
So loop runs **$\mathbf{O(n)}$** times, complexity:
$O(n) \times O(1) = O(n)$

Loop body:
$O(1)$

What's the complexity?

Outer loop is
O($n \log n$)

```
void something(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 1; j <= a.length; j *= 2)
      // something taking O(1) time
}
```

Inner loop is
O($\log n$)

A loop running through $i = 1, 2, 4, \ldots, n$ runs O($\log n$) times!

`while` loops

```
long squareRoot(long n) {
  long i = 0;
  long j = n + 1;
  while(i + 1 != j) {
    long k = (i + j) / 2;
    if (k * k <= n)
      i = k;
    else
      j = k;
  }
  return i;
}
```

… and halves `j - i`, so $\mathbf{O(\log n)}$ iterations

Each iteration takes O(1) time

Summary: loops

Basic rule for complexity of loops:

- Number of iterations times complexity of body
- `for (int i = 0; i < n; i++)` : n iterations
- `for (int i = 1; i ≤ n; i *= 2)` : $O(\log n)$ iterations
- `while` loops: same rule, but can be trickier to count number of iterations

If the complexity of the body depends on the value of the loop counter:

- e.g. $O(i)$, where $0 \le i < n$
- round it up to $O(n)$!

Sequences of statements

What's the complexity here?
(Assume that the loop bodies are O(1))

```
...
for (int i = 0; i < n; i++) ...
for (int i = 1; i < n; i *= 2) ...
...
```

First loop: $\mathbf{O(}\boldsymbol{n}\mathbf{)}$

Second loop: $\mathbf{O(\log}\ \boldsymbol{n}\mathbf{)}$

Total: $O(n) + O(\log n) = \mathbf{O(}\boldsymbol{n}\mathbf{)}$

For sequences, *add* the complexities!

A familiar scene

Outer loop:
n iterations, $O(n)$ body,
so: $\mathbf{O(n^2)}$

Rest of loop body $\mathbf{O(1)}$,
so loop body:
$O(1) + O(n) = \mathbf{O(n)}$

```
int[] array = {};
for (int i = 0; i < n; i++) {
  int[] newArray = new int[array.length+1];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  array = newArray;
}
```

Inner loop:
$\mathbf{O(n)}$

A familiar scene, take 2

Outer loop:
n/100 iterations, which is O(n)
with body O(n),
so still: **O(n^2)**

```
int[] array = {};
for (int i = 0; i < n; i += 100) {
  int[] newArray = new int[array.length+100];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  array = newArray;
}
```

A familiar scene, take 3

Outer loop:
$\log n$ iterations,
with body $O(n)$,
so: $\mathbf{O(n \log n)}$???

```
int[] array = {};
for (int i = 0; i < n; i *= 2) {
  int[] newArray = new int[array.length*2];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  array = newArray;
}
```

Inner loop:
$O(n)$

A familiar scene, take 3

```
int[] array = {};
for (int i = 0; i < n; i *= 2) {
  int[] newArray = new int[array.length*2];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  array = newArray;
}
```

Here we "round up" $O(i)$ to $O(n)$. This causes an overestimate!

A complication

Our algorithm has $O(n)$ complexity, but we've calculated $O(n \log n)$

- An overestimate, but not a severe one (If $n = 1000000$ then $n \log n = 20\mathrm{n}$)
- This can happen but is normally not severe
- To get the accurate answer: do the maths

Good news: for "normal" loops this doesn't happen

- If all bounds are n, or n^2, or another loop variable, or a loop variable squared, or …

Main exception: loop variable i doubles every time, body complexity depends on i

Doing the sums

In our example:

- The inner loop's complexity is $O(i)$
- In the outer loop, i ranges over $1, 2, 4, 8, ..., 2^a$

Instead of rounding up, we will add up the time for all the iterations of the loop:

- $1 + 2 + 4 + 8 + \ldots + 2^a$
  $= 2^{a+1} - 1 < 2 \times 2^a$

Since $2^a \le n$, the total time is at most $2n$, which is $O(n)$

A last example

The outer loop runs $O(\log n)$ times

The `j`-loop runs n^2 times

```
for (int i = 1; i <= n; i *= 2) {
  for (int j = 0; j < n*n; j++)
    for (int k = 0; k <= j; k++)
      // O(1)
  for (int j = 0; j < n; j++)
    // O(1)
}
```

`k <= j < n*n`
so his loop is:
$O(n^2)$

This loop is:
$O(n)$

Total: $O(\log n) \times (O(n^2) \times O(n^2) + O(n))$
$= O(n^4 \log n)$

Summary big-O

Big-O complexity:

- Calculate runtime without doing hard sums!
- Lots of “rules of thumb” that work almost all of the time
- Very occasionally, still need to do hard sums :(
- Ignoring constant factors: seems to be a good tradeoff

Complexity of recursive functions

Calculating complexity

Let $\mathrm{T}(n)$ be the time that `f` takes on a list of size n

```
f :: [a] -> [a]
f []  = []
f [x] = [x]
f xs  = g (f ys) (f zs)
 where
  (ys, zs) = splitInTwo xs
```

Two recursive calls of size $n/2$

Assume $\mathrm{O}(g) = \mathrm{O}(n)$ then $\mathrm{T}(n) = \mathrm{O}(n) + 2\mathrm{T}(n/2)$

Recurrence relations

Procedure for calculating complexity of a recursive algorithm:

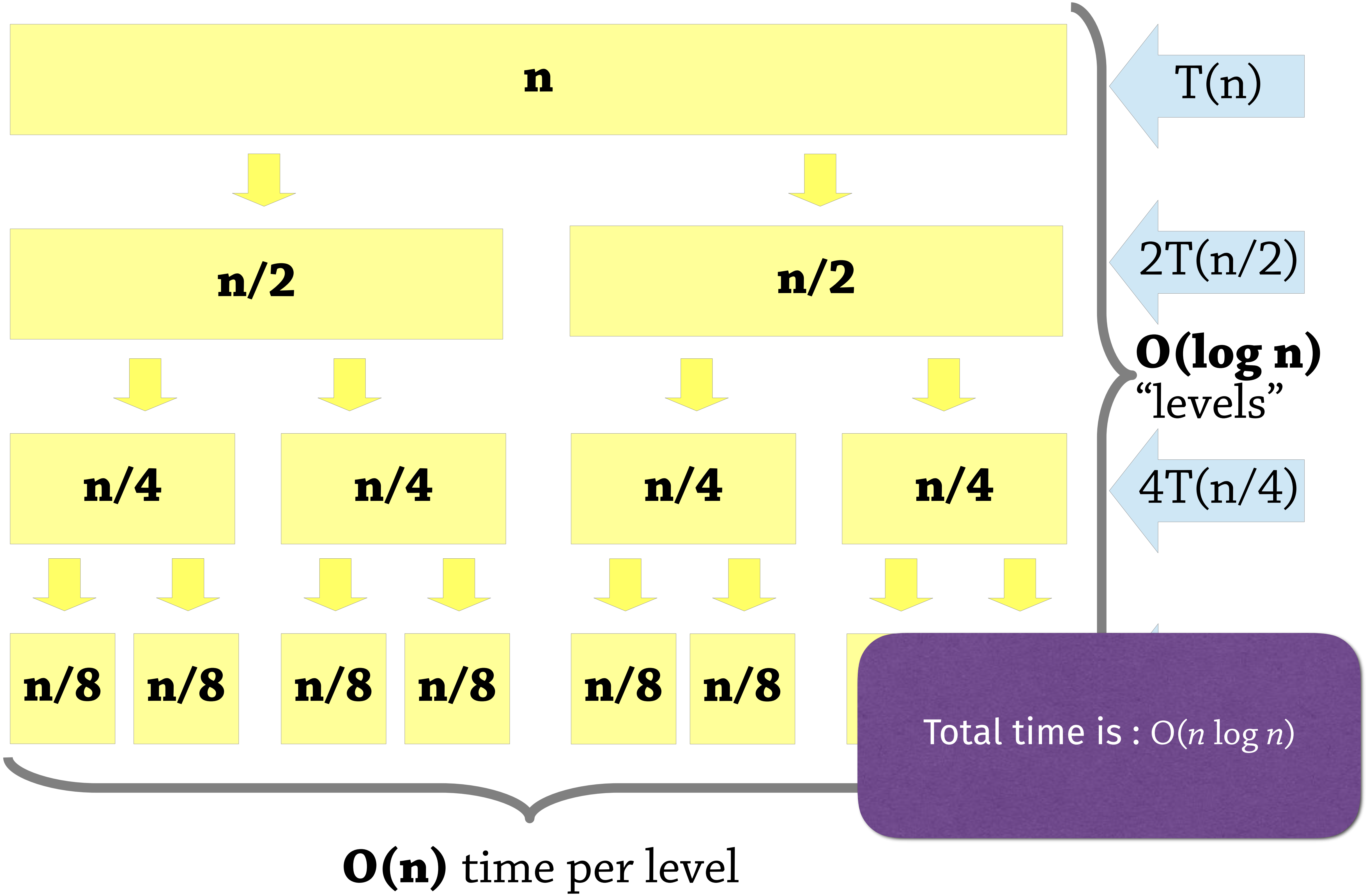
- Write down a *recurrence relation*
  e.g. $T(n) = O(n) + 2T(n/2)$

- Solve the recurrence relation to get a formula for $T(n)$ (difficult!)

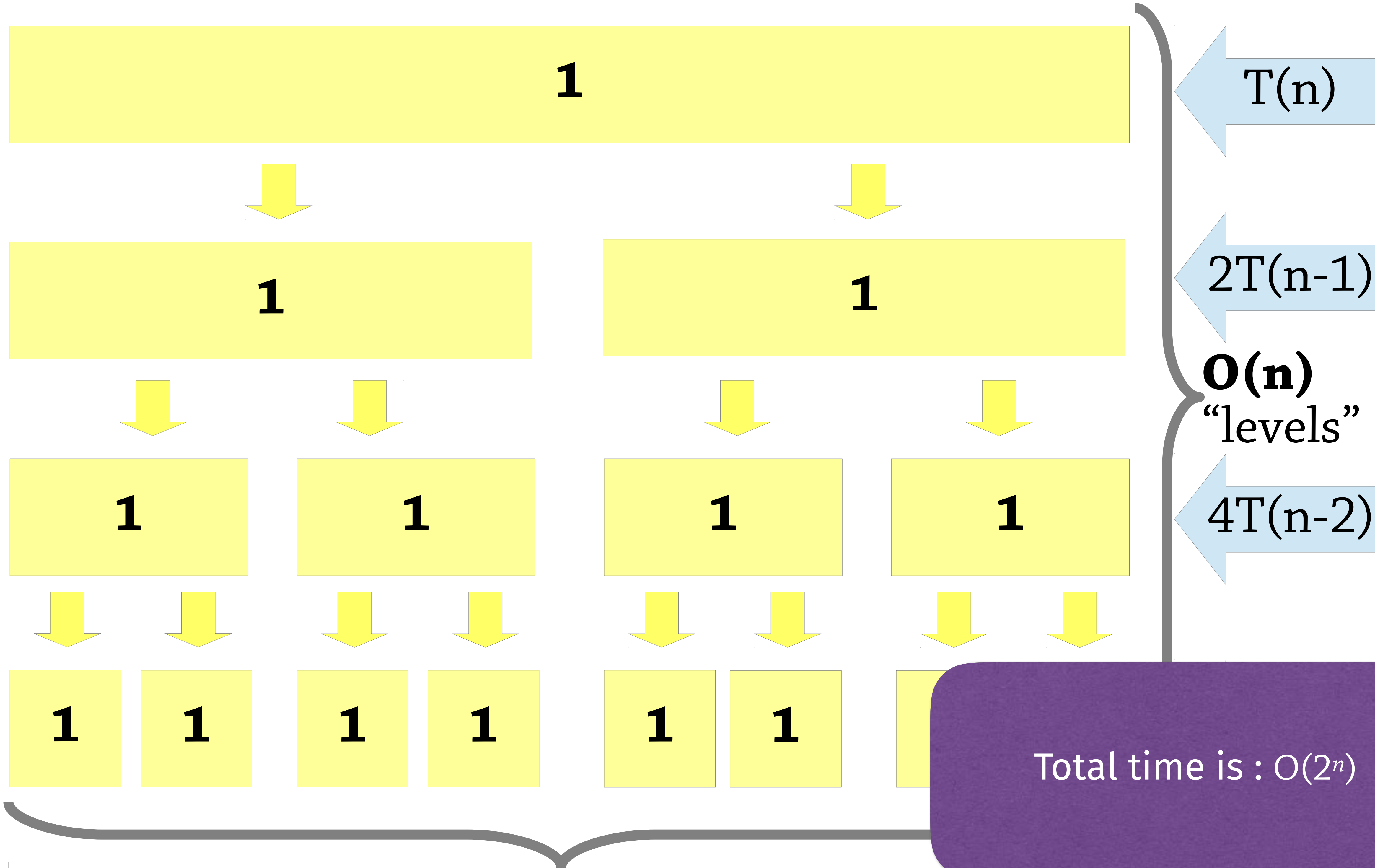
There isn't a general way of solving any recurrence relation – we'll just see a few families of them

First approach: draw a diagram

Draw a diagram



Another example: $T(n) = O(1) + 2T(n\text{-}1)$



Total time is : $O(2^n)$

amount of work **doubles** at each level

This approach

- Good for building an intuition
- Maybe a bit error-prone

- Second approach: *expand out* the definition
- Example: solving $T(n) = O(1) + T(n-1)$

Expanding out recurrence relations

$T(n) = 1 + T(n\text{-}1)$ $\{ T(n\text{-}1) = 1 + T(n\text{-}2) \}$

$= 1 + 1 + T(n\text{-}2) = 2 + T(n\text{-}2)$

$= 3 + T(n\text{-}3)$

$= \ldots$

$= n + T(0)$

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

$= O(n)$

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

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

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

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

$= \ldots$

$= n + n/2 + n/4 + n/8 \ldots = n + n\,(1/2 + 1/4 + 1/8 + \ldots) = n + {\sim}n$

$< 2n$

$= O(n)$

Functions that recurse once

$T(n) = O(1) + T(n\text{-}1)$: $T(n) = O(n)$

$T(n) = O(n) + T(n\text{-}1)$: $T(n) = O(n^2)$

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

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

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

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

- for example our function `f` (this is mergesort!)

$T(n) = O(1) + 2T(n\text{-}1)$: $T(n) = O(2^n)$

- Because 2^n recursive calls of depth n

Other cases: *master theorem* (see Wikipedia)

- Beyond the scope of this course

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 can help!

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