095946 ADVANCED ALGORITHMS AND PARALLEL PROGRAMMING

Fabrizio Ferrandi

a.a. 2021-2022

- Material adapted from Erik D. Demaine and Charles E. Leiserson slides

Algorithm as a recipe

Ingredients:

- spaghetti:450g (1 pound)
- bacon: 225g (½ pound)
- egg yolks: 5
- Pecorino or Parmigiano-Reggiano cheese, grated: 360ml (1½ cups)
- olive oil, extra-virgin: 3-4 tablespoons
- pepper, freshly ground: ½ tablespoon
- Salt

Utensils:

- large pot
- large skillet
- bowl
- measuring cups and spoons
- fork

Procedure

1. Dice the bacon into small pieces (1 inch [2.5cm] will do).
2. The Pot: Bring a big pot of water to a boil and add salt when it begins to simmer.
3. The Pot: Cook the spaghetti until it is al dente and drain it, reserving ½ cup (118 ml) of water.
4. The Skillet: As spaghetti is cooking, heat the olive oil in a large skillet over a medium-high heat. When the oil is hot, add the pancetta and cook for about 10 minutes over a low flame until the pancetta has rendered most of its fat but is still chewy and barely browned.
5. The Bowl: In a bowl, slowly whisk about ½ cup of the pasta water into the egg yolks, using a fork. Add the Parmesan cheese and pepper. Mix with a fork.
6. The Skillet: Transfer the spaghetti immediately to the skillet with the pancetta. Toss it and turn off the heat. Add the egg mixture to the skillet with the pasta and toss all the ingredients to coat the pasta. Taste the pasta and add salt and black pepper, if necessary.

Algorithm as a recipe

How long it takes to prepare *carbonara* for 2 persons?

- How long for 10
  - How long for 100

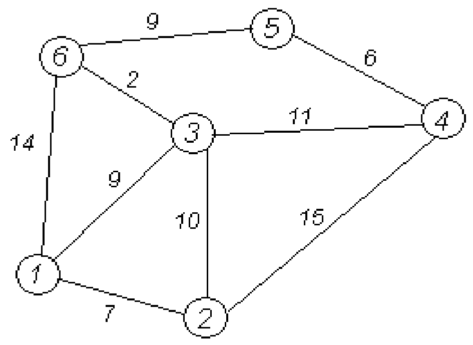
What happens if a friend helps you?

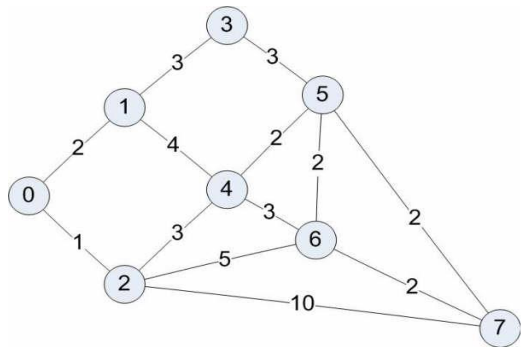
- What happens if 10 friends helps you?
  - What happens if 100 friends helps you?

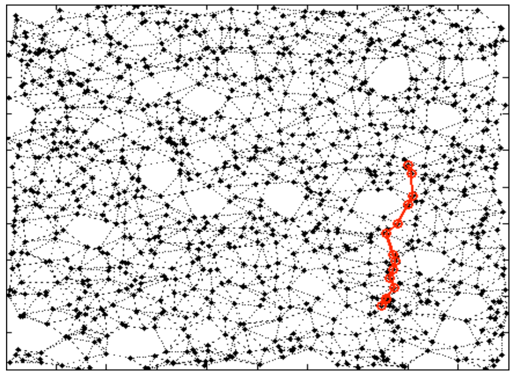
Algorithm

- An **algorithm** is any **well-defined** computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**.
- It must terminate in a finite number of steps

Shortest path computation







Is there exist an algorithm computing the shortest path between two vertices in a graph?
How long does it take?
May parallelization help?

Analysis of algorithms

The theoretical study of computer-program performance and resource usage.

What's more important than performance?

- modularity
- correctness
- maintainability
- functionality
- robustness
- user-friendliness
- programmer time
- simplicity
- extensibility
- reliability

Why study algorithms and performance?

Algorithms help us to understand *scalability*.

Performance often draws the line between what is feasible and what is impossible.

Algorithmic mathematics provides a *language* for talking about program behavior.

Performance is the *currency* of computing.

The lessons of program performance generalize to other computing resources.

Speed is fun!

The problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example:

Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Insertion sort

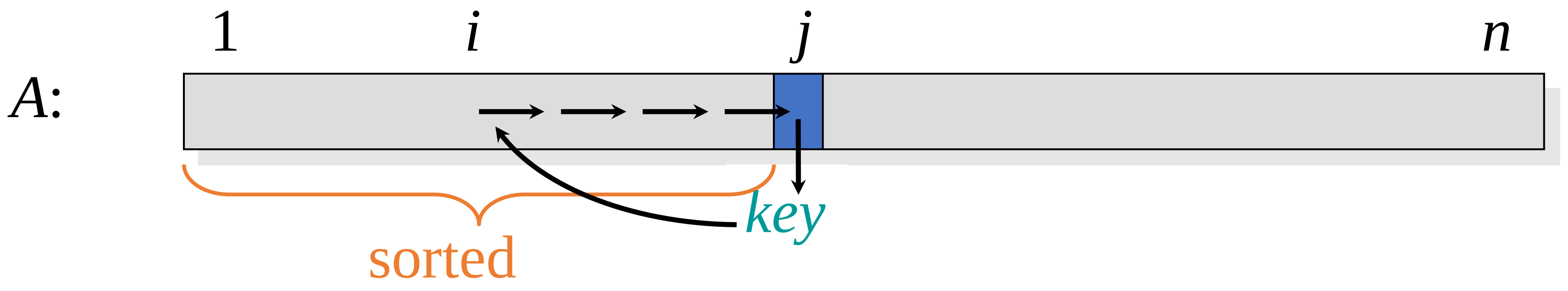
"pseudocode"

```
INSERTION-SORT $(A, n)$        ⊳ $A[1 \,.\,.\, n]$
    for $j \leftarrow 2$ to $n$
            do $key \leftarrow A[j]$
                $i \leftarrow j - 1$
                while $i > 0$ and $A[i] > key$
                        do $A[i+1] \leftarrow A[i]$
                            $i \leftarrow i - 1$
                $A[i+1] = key$
```

Insertion sort

“pseudocode”

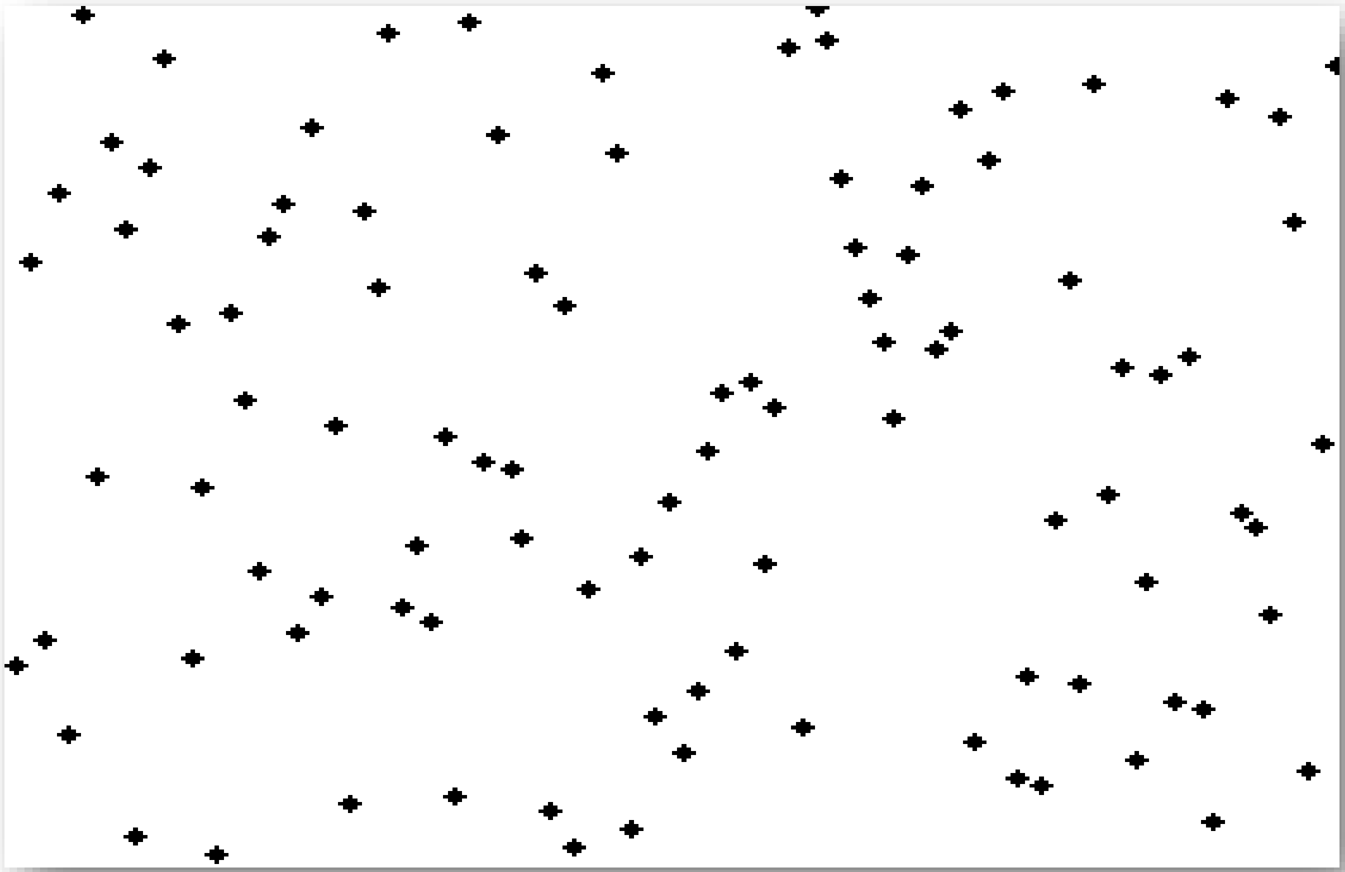
```
INSERTION-SORT $(A, n)$        $\triangleright A[1 \,..\, n]$
    for $j \leftarrow 2$ to $n$
        do $key \leftarrow A[j]$
           $i \leftarrow j - 1$
           while $i > 0$ and $A[i] > key$
               do $A[i+1] \leftarrow A[i]$
                  $i \leftarrow i - 1$
           $A[i+1] = key$
```



Insertion Sort C implementation?

```
void insertion_sort(std::vector<int> &A)
{
  int i;
  int j;
  int value;
  for(i=1; i<A.size(); i++)
  {
    value = A[i];
    j = i-1;
    while(j>=0 && A[j]>value)
    {
      A[j+1] = A[j];
      j = j-1;
    }
    A[j+1] = value;
  }
}
```

Insertion Sort animation from Wikipedia


Insertion Sort animation from Wikipedia

http://en.wikipedia.org/wiki/Insertion_sort

http://en.wikipedia.org/wiki/Image:Insertion_sort_animation.gif

Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Kinds of analyses

Worst-case: (usually)

- $T(n)$ = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (bogus)

- Cheat with a slow algorithm that works fast on *some* input.

Machine-independent time

What is insertion sort's worst-case time?

- It depends on the speed of our computer:
    - relative speed (on the same machine),
    - absolute speed (on different machines).

BIG IDEA:

- Ignore machine-dependent constants.
- Look at *growth* of $T(n)$ as $n \rightarrow \infty$.

"Asymptotic Analysis"

Θ-notation

Math:

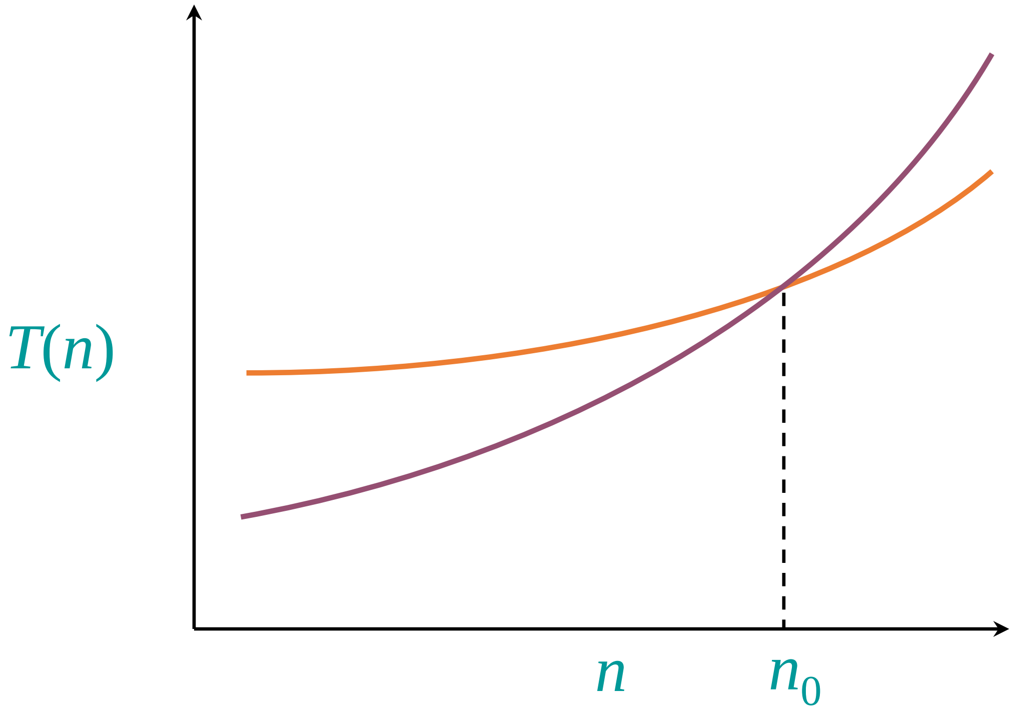
$\Theta(g(n)) = \{ f(n) :$ there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$

Engineering:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

Asymptotic performance

When n gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.



- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.

Set definition of O-notation

$O(g(n)) = \{ f(n) :$ there exist constants $c > 0, n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$

EXAMPLE: $2n^2 \in O(n^3)$

Ω-notation (lower bounds)

O-notation is an *upper-bound* notation. It makes no sense to say $f(n)$ is at least $O(n^2)$.

$\Omega(g(n)) = \{\ f(n) :$ there exist constants $c > 0,\ n_0 > 0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\ \}$

EXAMPLE: $\sqrt{n} = \Omega(\lg n)$ $(c = 1,\ n_0 = 16)$

Θ-notation (tight bounds)

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

EXAMPLE: $$\frac{1}{2}n^2 - 2n = \Theta(n^2)$$

ο-notation and ω-notation

O-notation and Ω-notation are like ≤ and ≥.
o-notation and ω-notation are like < and >.

$o(g(n)) = \{ f(n)$: for any constant $c > 0$,
there is a constant $n_0 > 0$ such that $0 \leq f(n) <$
$cg(n)$ for all $n \geq n_0$ $\}$

EXAMPLE: *2n2 = o(n3)*

o-notation and ω-notation

O-notation and Ω-notation are like $\leq$ and $\geq$.
o-notation and ω-notation are like $<$ and $>$.

$\omega(g(n)) = \{ f(n)$: for any constant $c > 0$,
there is a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$ $\}$

EXAMPLE: $\sqrt{n} = \omega(\lg n)$ $(n_0 = 1+1/c)$

Insertion sort analysis

Worst case: Input reverse sorted.

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$ [arithmetic series]

Average case: All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

Is insertion sort a fast sorting algorithm?

- Moderately so, for small n.
- Not at all, for large n.

Merge sort

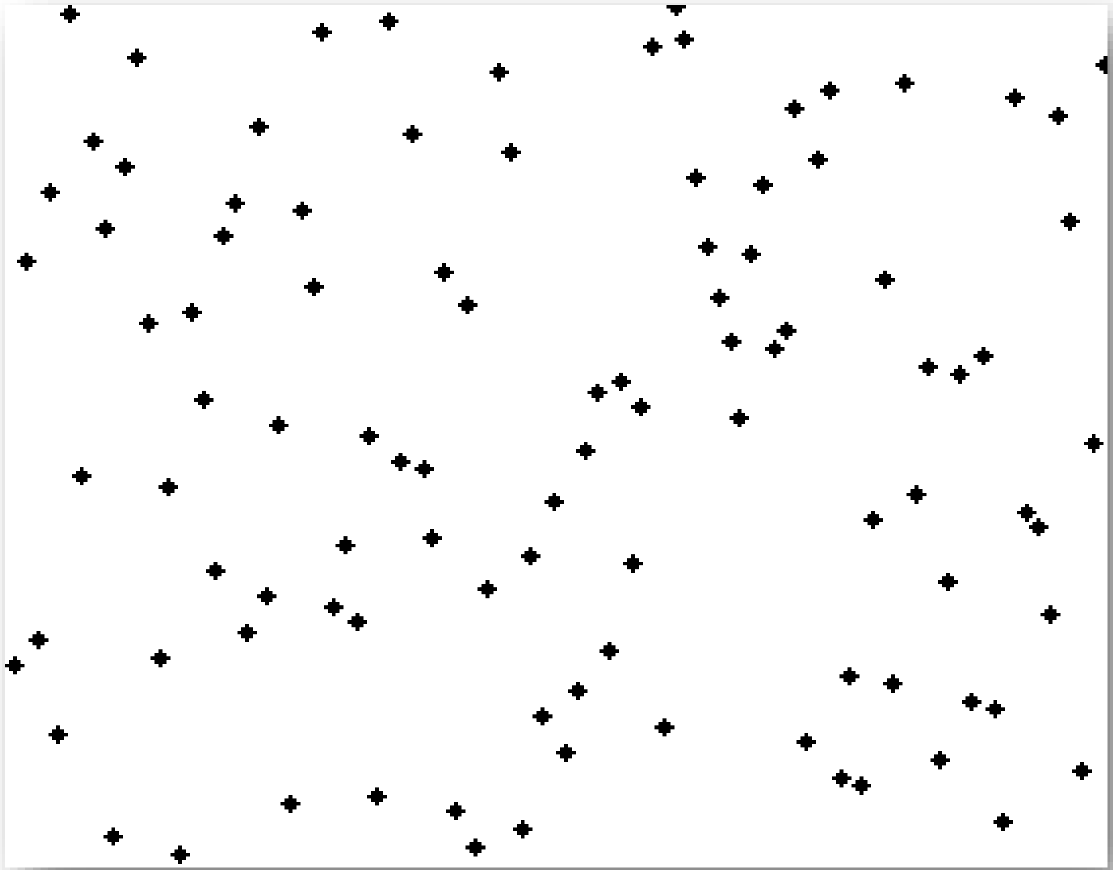
```
MERGE-SORT $A[1\,..\,n]$
    1. If $n = 1$, done.
    2. Recursively sort $A[\,1\,..\,\lceil n/2\rceil\,]$
       and $A[\,\lceil n/2\rceil+1\,..\,n\,]$ .
    3. "*Merge*" the 2 sorted lists.
```

Key subroutine: MERGE

Merge Sort in C++

```
void merge_sort(std::vector<int> &A)
{
          if (A.size()==1) return;
          std::vector<int> A1;
          std::vector<int> A2;
          for(int i=0; i<A.size()/2; i++)
                    A1.push_back(A[i]);
          for(int i=A.size()/2; i<A.size(); i++)
                    A2.push_back(A[i]);
          merge_sort(A1);
          merge_sort(A2);
          A = merge(A1,A2);
}
```

Merge Sort: animation from Wikipedia


Merge Sort animation from Wikipedia

http://en.wikipedia.org/wiki/Merge_sort

http://upload.wikimedia.org/wikipedia/en/c/c5/Merge_sort_animation2.gif

Analyzing merge sort

$T(n)$

$\Theta(1)$

$2T(n/2)$

Abuse

$\Theta(n)$

MERGE-SORT $A[1\,.\,.\,n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1\,.\,.\lceil n/2\rceil\,]$ and $A[\lceil n/2\rceil+1\,.\,.\,n\,]$.
3. *"Merge"* the 2 sorted lists

Sloppiness: Should be $T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor)$, but it turns out not to matter asymptotically.

Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$T(n)$$

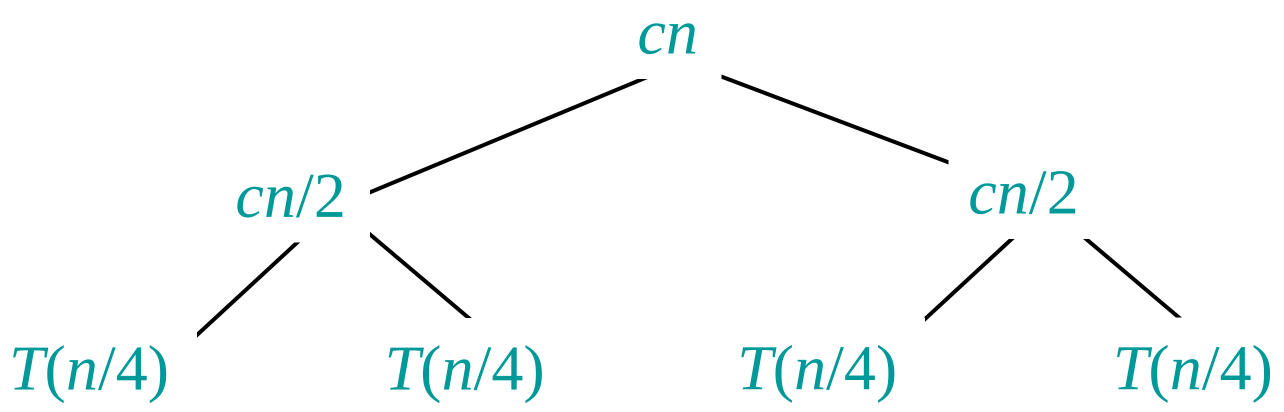
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$\begin{tikzcd}
 & cn \arrow[dl, no head] \arrow[dr, no head] & \\
T(n/2) & & T(n/2)
\end{tikzcd}$$

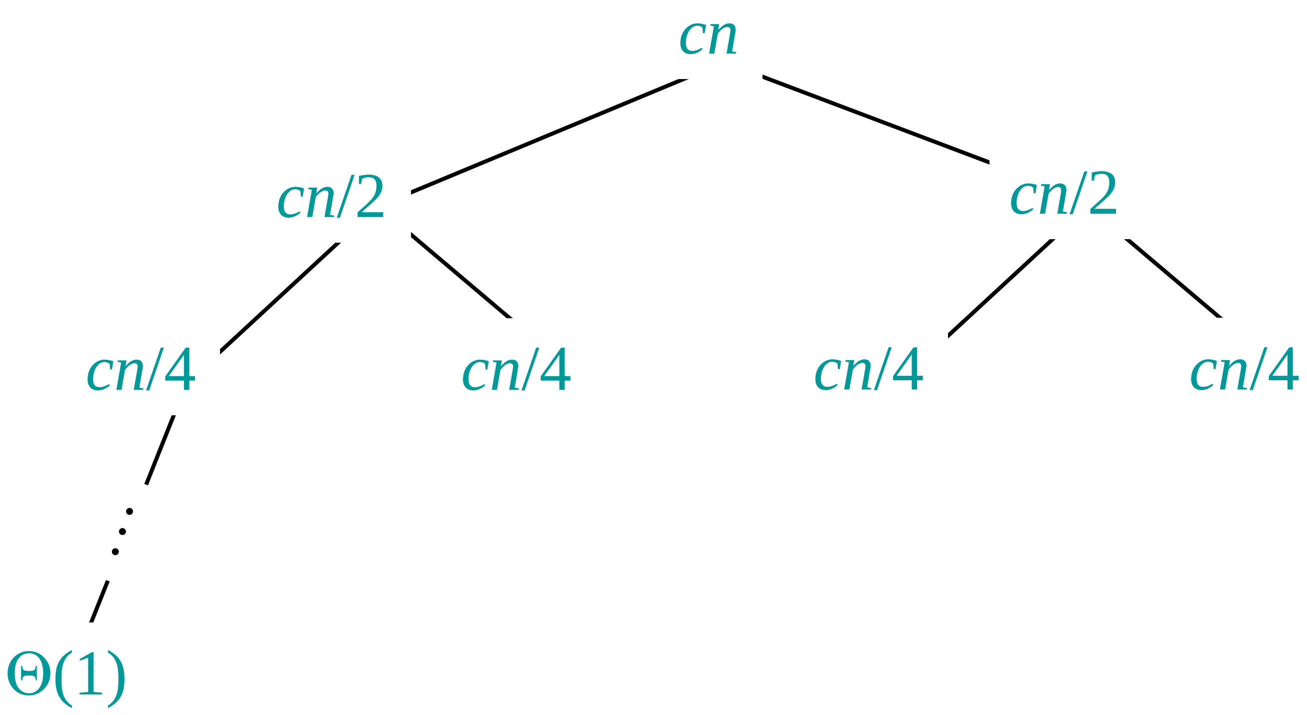
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



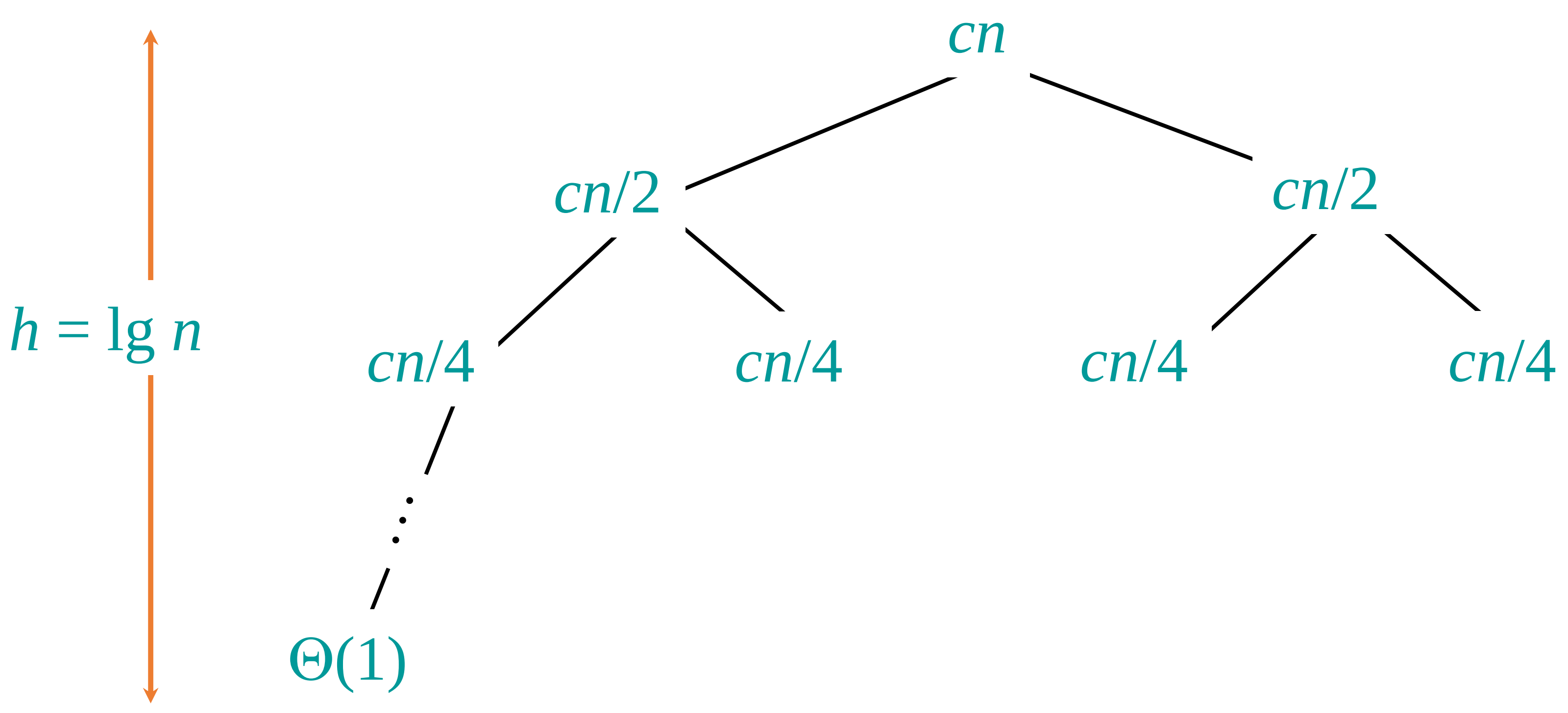
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



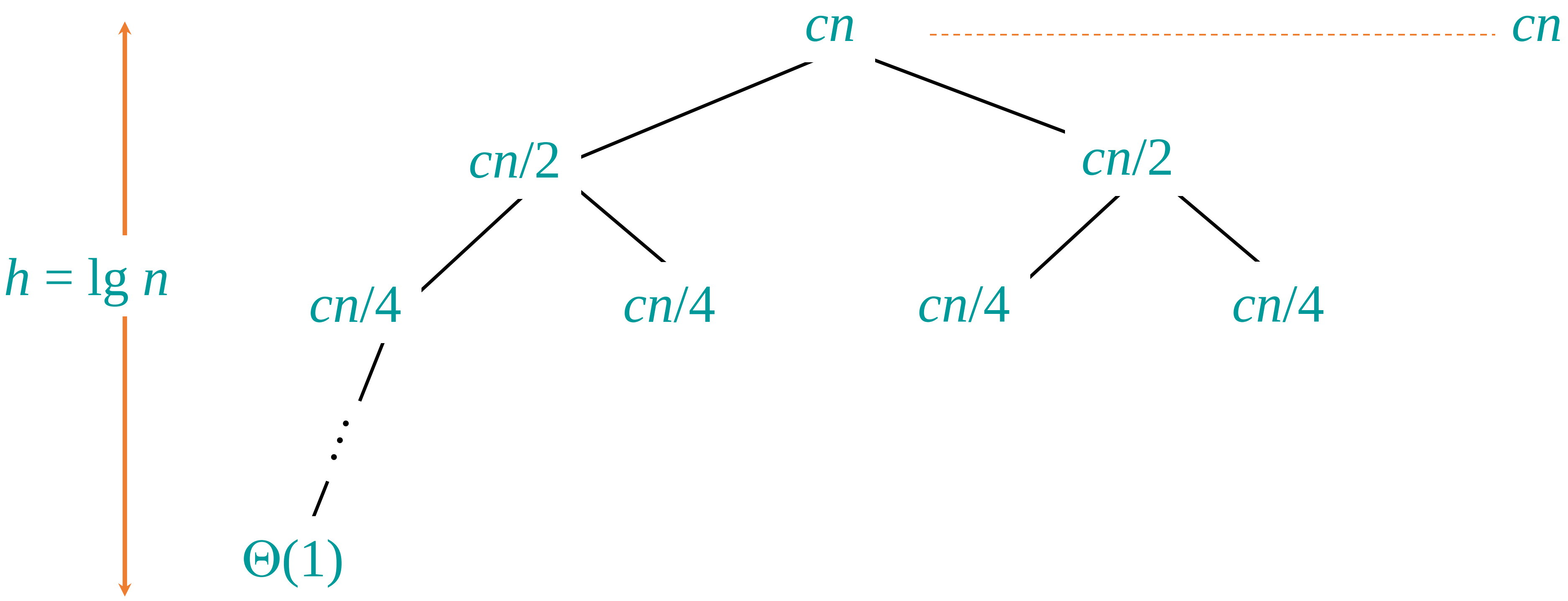
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



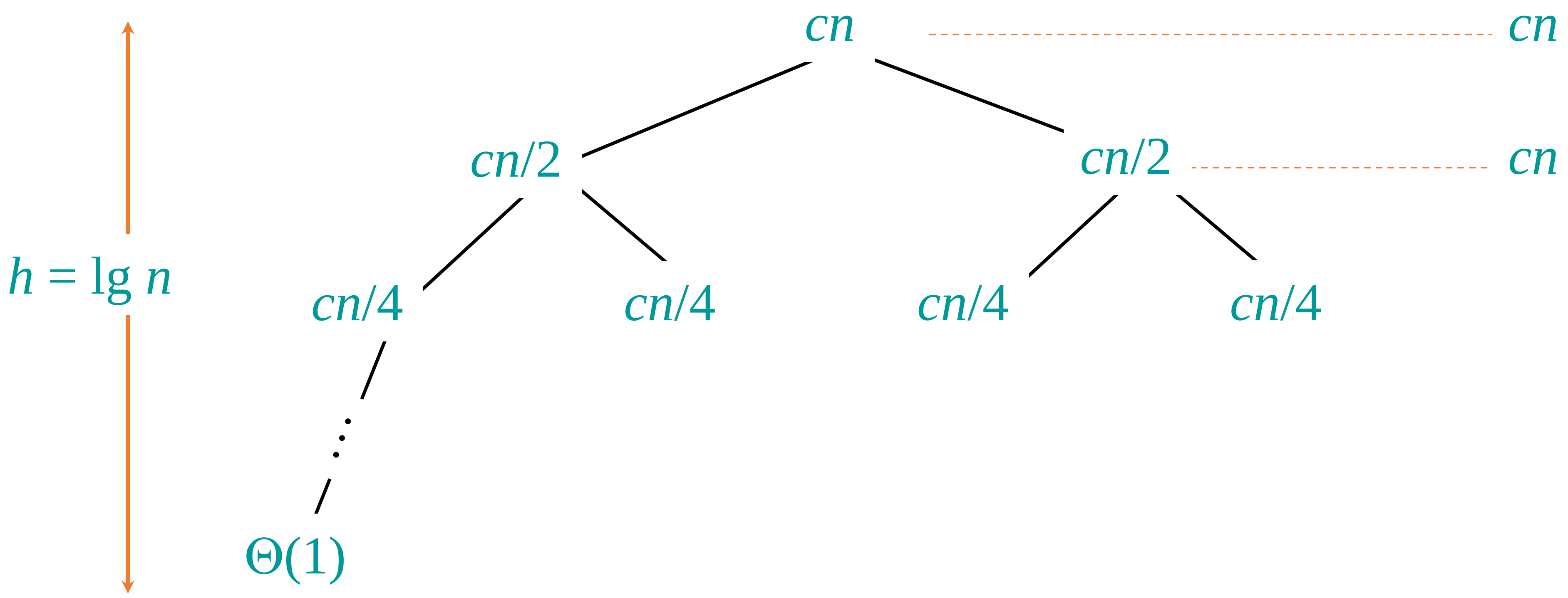
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



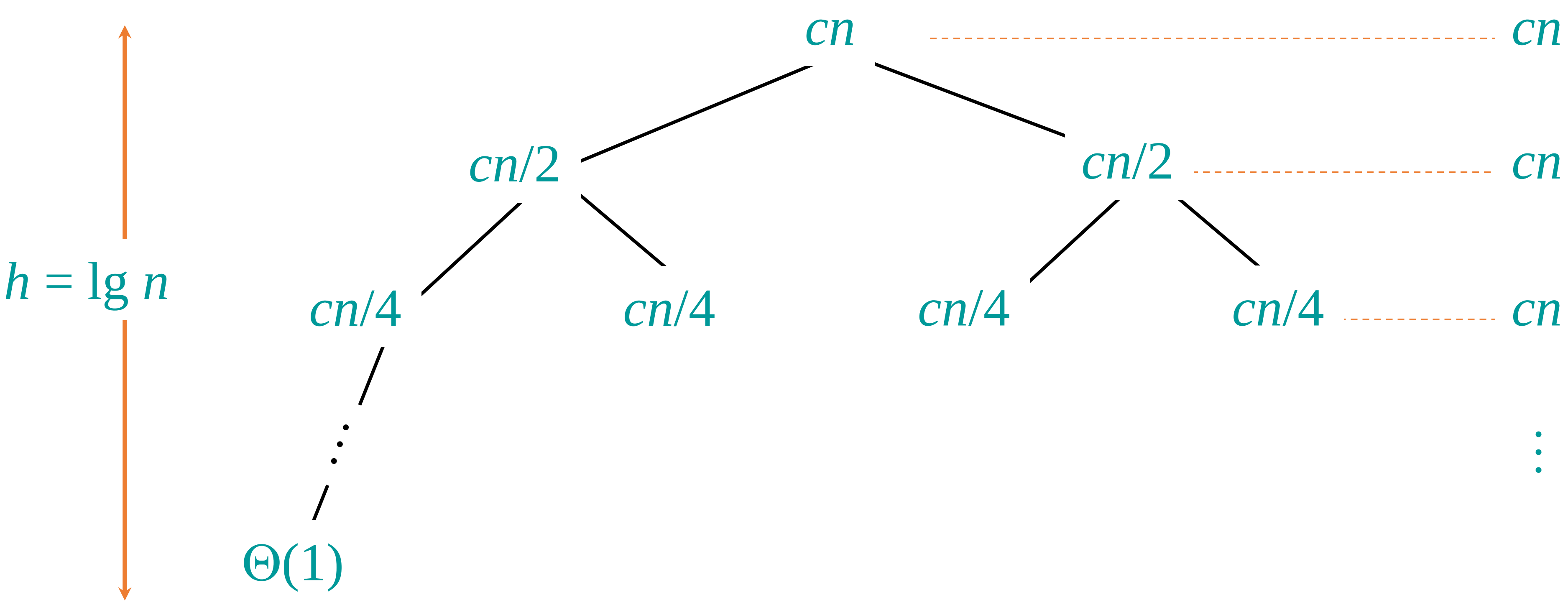
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



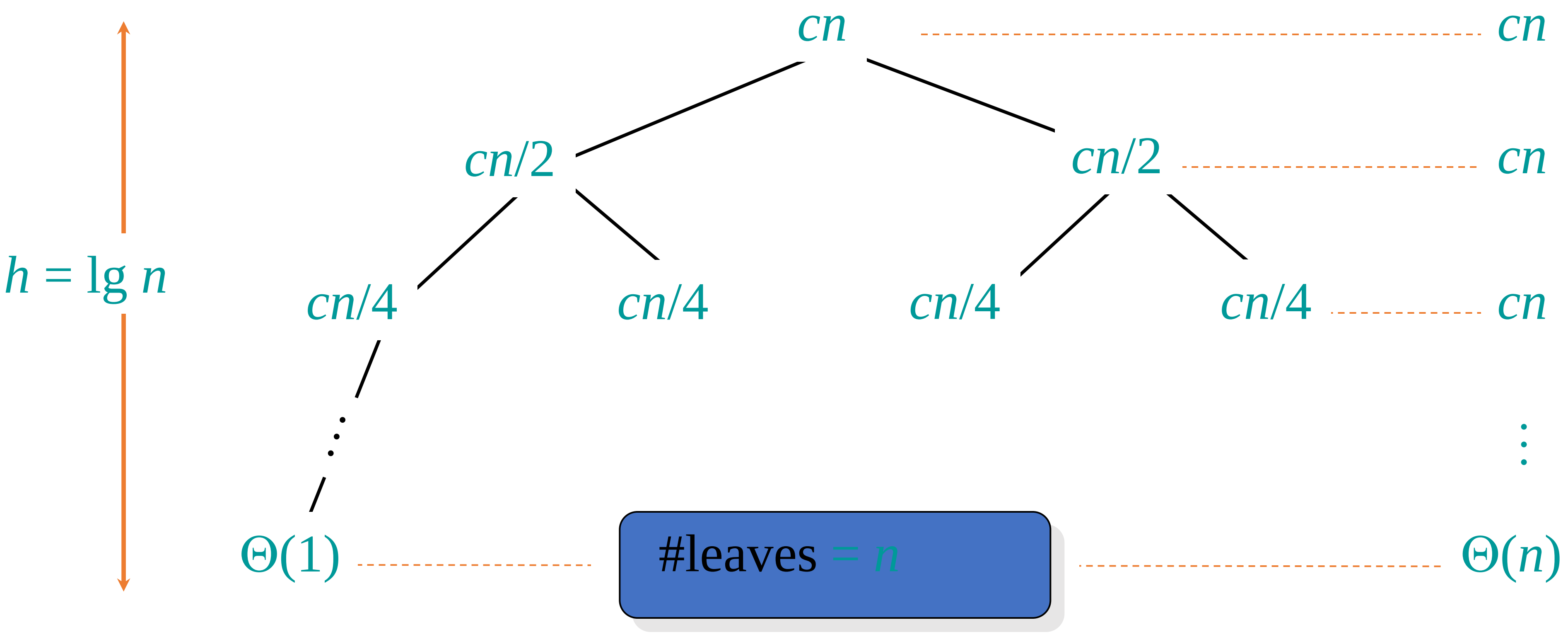
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



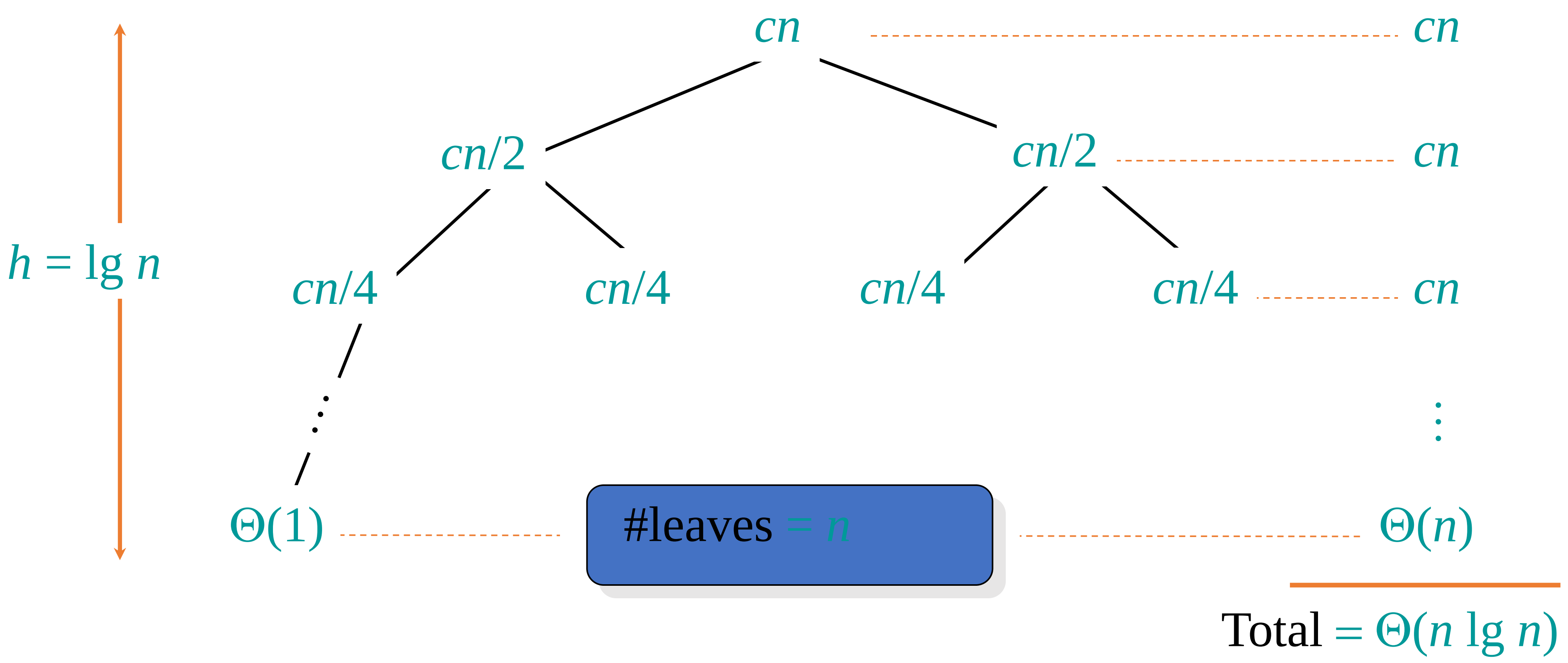
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



First intuition

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.
- Go test it out for yourself!

Solving recurrences

Substitution method

The most general method:

1. *Guess* the form of the solution.
2. *Verify* by induction.
3. *Solve* for constants.

Substitution method

The most general method:

1. *Guess* the form of the solution.
2. *Verify* by induction.
3. *Solve* for constants.

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

- [Assume that $T(1) = \Theta(1)$.]
- Guess $O(n^3)$. (Prove O and Ω separately.)
- Assume that $T(k) \leq ck^3$ for $k < n$.
- Prove $T(n) \leq cn^3$ by induction.

The master method

The master method applies to recurrences of the form

asymptotic analysis for recursion

$$T(n) = a\,T(n/b) + f(n)\ ,$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

Three common cases

Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

   Solution: $T(n) = \Theta(n^{\log_b a})$.

Three common cases

Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

   Solution: $T(n) = \Theta(n^{\log_b a})$.

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Three common cases (cont.)

Compare $f(n)$ with $n^{\log_b a}$:

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

   and $f(n)$ satisfies the *regularity condition* that $a f(n/b) \le c f(n)$ for some constant $c < 1$.

   Solution: $T(n) = \Theta(f(n))$.

Examples

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

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n.$

CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

$\therefore\ T(n) = \Theta(n^2).$

Examples

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

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n.$

CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

$\therefore\ T(n) = \Theta(n^2)$.

Ex. $T(n) = 4T(n/2) + n^2$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2.$

CASE 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.

$\therefore\ T(n) = \Theta(n^2 \lg n)$.

Examples

Ex. $T(n) = 4T(n/2) + n^3$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^3.$

CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(n/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore\ T(n) = \Theta(n^3).$

Examples

Ex. $T(n) = 4T(n/2) + n^3$
$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^3.$
CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$
and $4(n/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.
$\therefore T(n) = \Theta(n^3)$.

Ex. $T(n) = 4T(n/2) + n^2/\lg n$
$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2/\lg n.$
Master method does not apply.
In particular, for every constant $\varepsilon > 0$, we have
$n^{\varepsilon} = \omega(\lg n)$.