CS180: Algorithms and Complexity

Professor: Raghu Meka (raghum@cs)

Plan for Today

Master theorem

Integer multiplication

Exponentiation

Asymptotic analysis

Methodology for comparing run-times

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \rightarrow \mathbb{R}^{+}$

f(n) = O(g(n)):	iff there is a constant c>0 so that f(n) is eventually-always **at most** c g(n)

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

f(n) = O(g(n)): iff there is a constant c>0 so that
f(n) is eventually-always **at most** c g(n)

f(n) = Ω(g(n)): iff there is a constant c>0 so that
f(n) is eventually-always **at least** c g(n)

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

f(n) = O(g(n)): iff there is a constant c>0 so that f(n) is eventually-always **at most** c g(n)

f(n) = Ω(g(n)): iff there is a constant c>0 so that f(n) is eventually-always **at least** c g(n)

f(n) = Θ(g(n)): iff both hold - there are constants c_1, c_2>0 so that eventually always $c_1 g(n) < f(n) < c_2 g(n)$

Asymptotic analysis

Methodology for comparing run-times

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

f(n) = o(g(n)): f(n)/g(n) tends to 0 as n goes to infinity.

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

f(n) = o(g(n)): f(n)/g(n) tends to 0 as n goes to infinity.

f(n) = ω(g(n)): f(n)/g(n) tends to infinity as n goes to infinity.

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

f(n) = o(g(n)): f(n)/g(n) tends to 0 as n goes to infinity.

f(n) = ω(g(n)): f(n)/g(n) tends to infinity as n goes to infinity.

$$Ex : f(n) = n, g(n) = n^2.$$

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

f(n) = o(g(n)): f(n)/g(n) tends to 0 as n goes to infinity.

f(n) = ω(g(n)): f(n)/g(n) tends to infinity as n goes to infinity.

$$Ex : f(n) = n, g(n) = n^2. \qquad f = o(g).$$

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

f(n) = o(g(n)): f(n)/g(n) tends to 0 as n goes to infinity.

f(n) = ω(g(n)): f(n)/g(n) tends to infinity as n goes to infinity.

$$Ex: f(n) = n, g(n) = n^2. \qquad f = o(g).$$

$$Ex: f(n) = n^3, g(n) = n^{2.9}.$$

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

f(n) = o(g(n)): f(n)/g(n) tends to 0 as n goes to infinity.

f(n) = ω(g(n)): f(n)/g(n) tends to infinity as n goes to infinity.

$$Ex : f(n) = n, g(n) = n^2. \qquad f = o(g).$$

$$Ex : f(n) = n^3, g(n) = n^{2.9}. \quad f = \omega(g).$$

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide problem into several subproblems.
- Solve each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Last class:

- Example for Mergesort
- Recursion analysis for mergesort

Complexity of mergesort

Defn: $T(n)$ = # Comparisons made by mergesort in worst-case on array with n elements.

Mergesort recurrence: $T(1) = 1$

Complexity of mergesort

Defn: T(n) = # Comparisons made by mergesort
in worst-case on array with n elements.

Mergesort recurrence: T(1) = 1

$$\boxed{T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}}$$

Complexity of mergesort

Defn: T(n) = # Comparisons made by mergesort in worst-case on array with n elements.

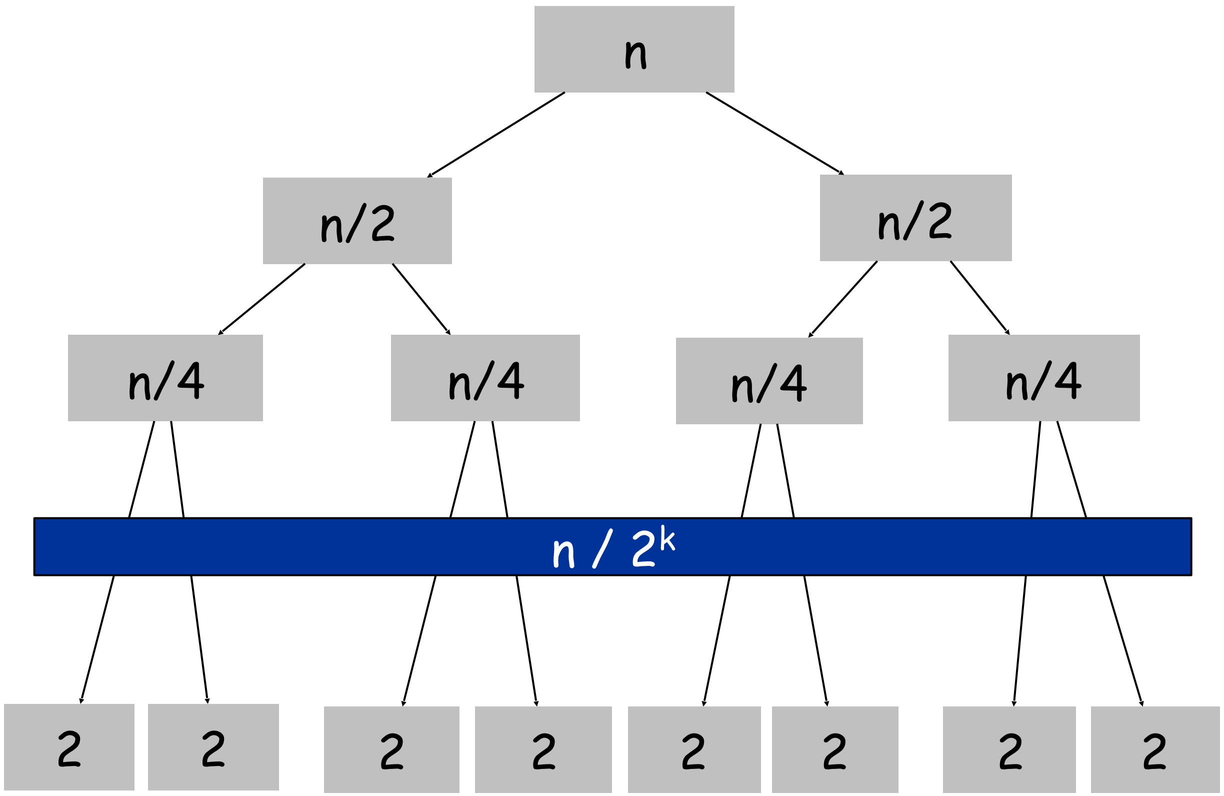
Mergesort recurrence: T(1) = 1

$$\boxed{T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}}$$

Solution: O(n log n)

Proof by recursion tree

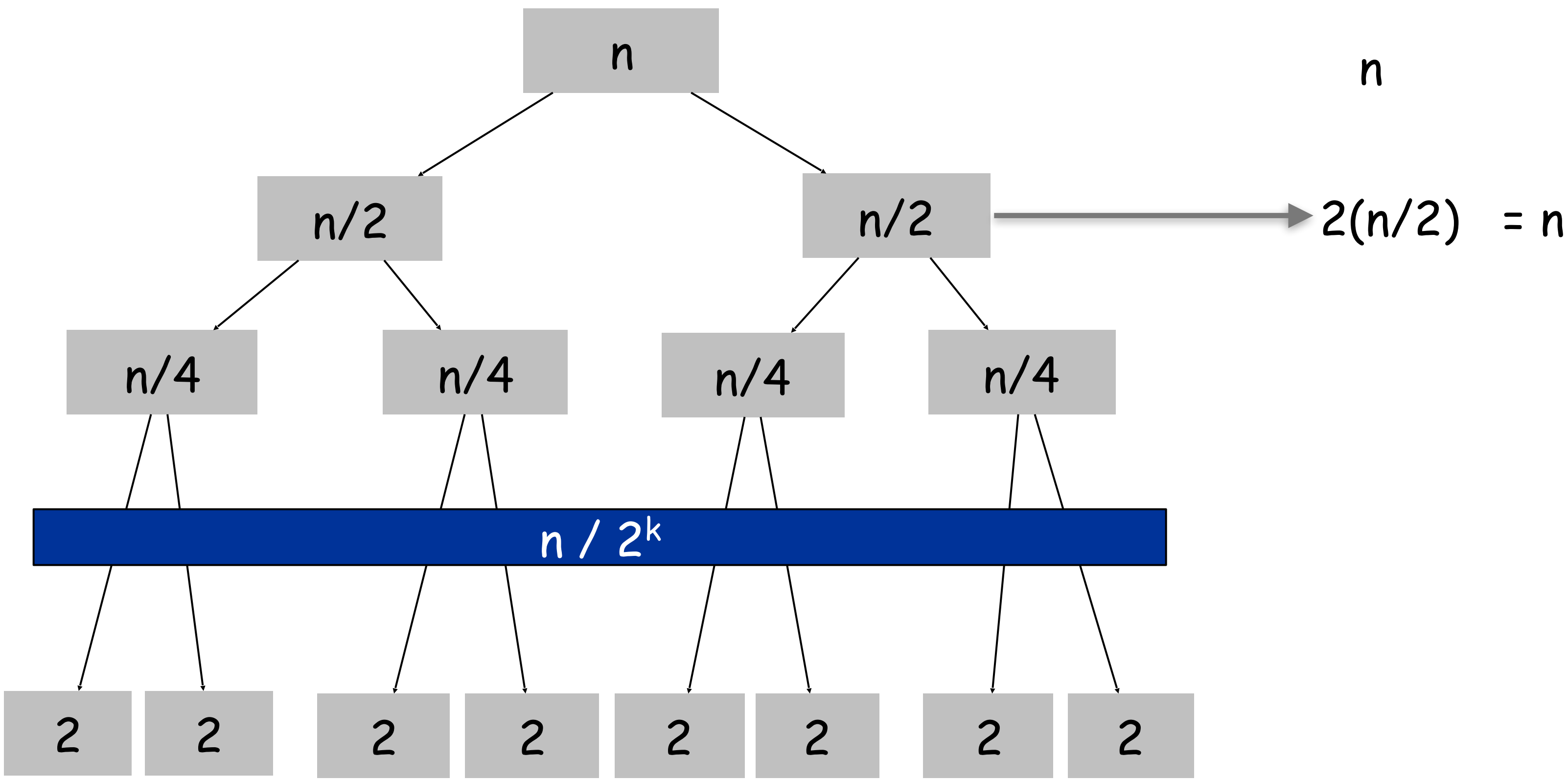
$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



n

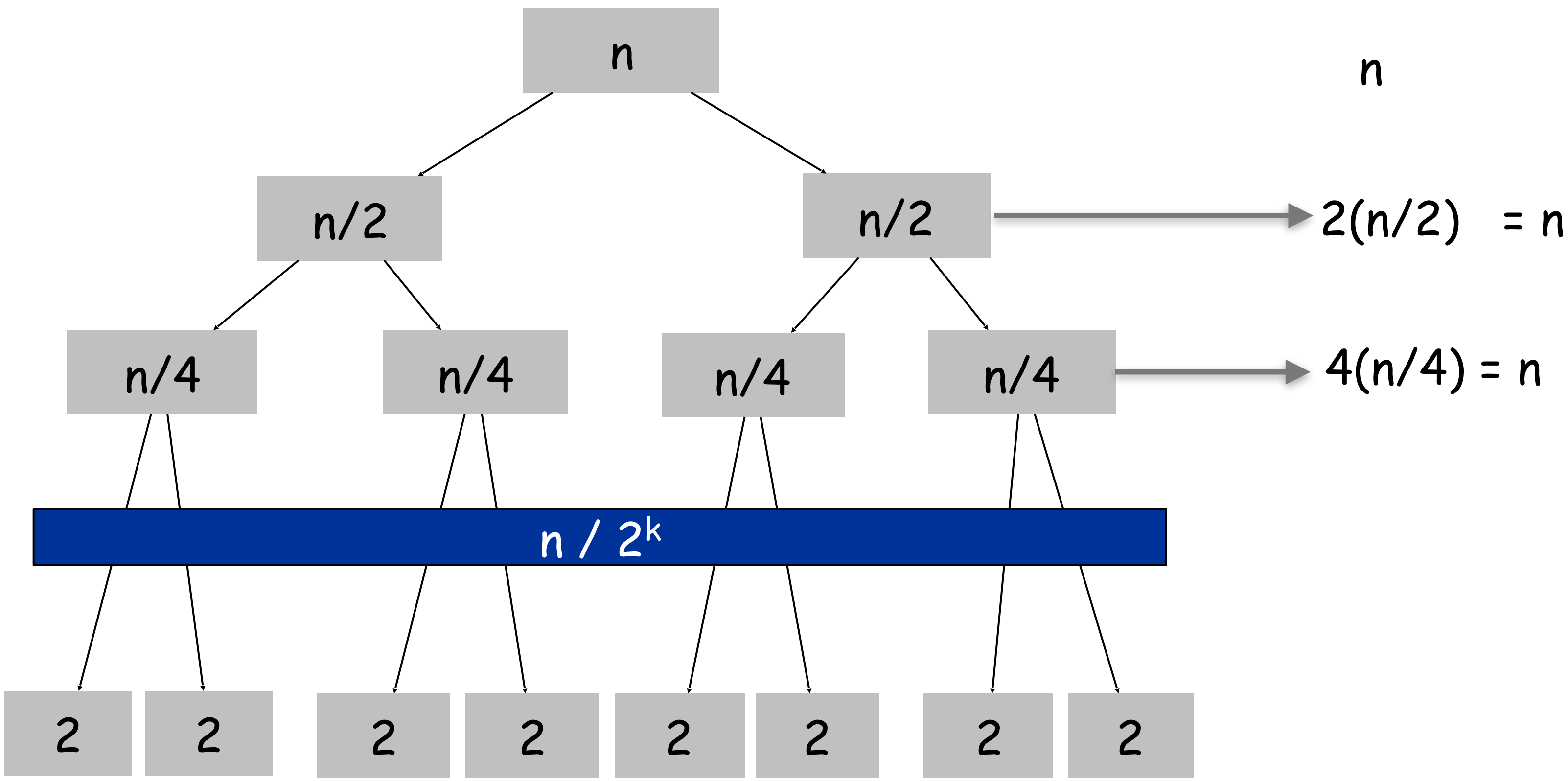
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



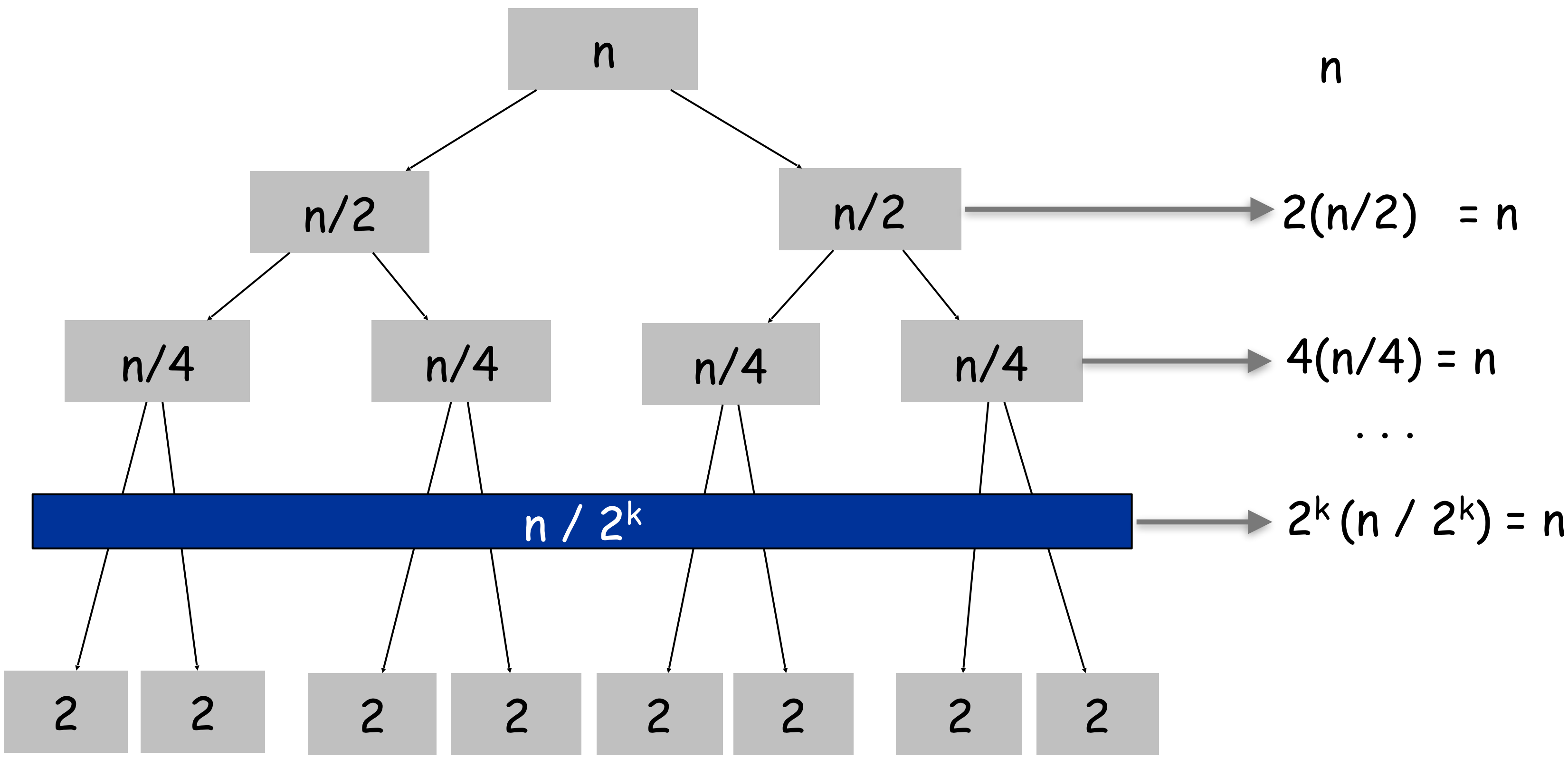
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



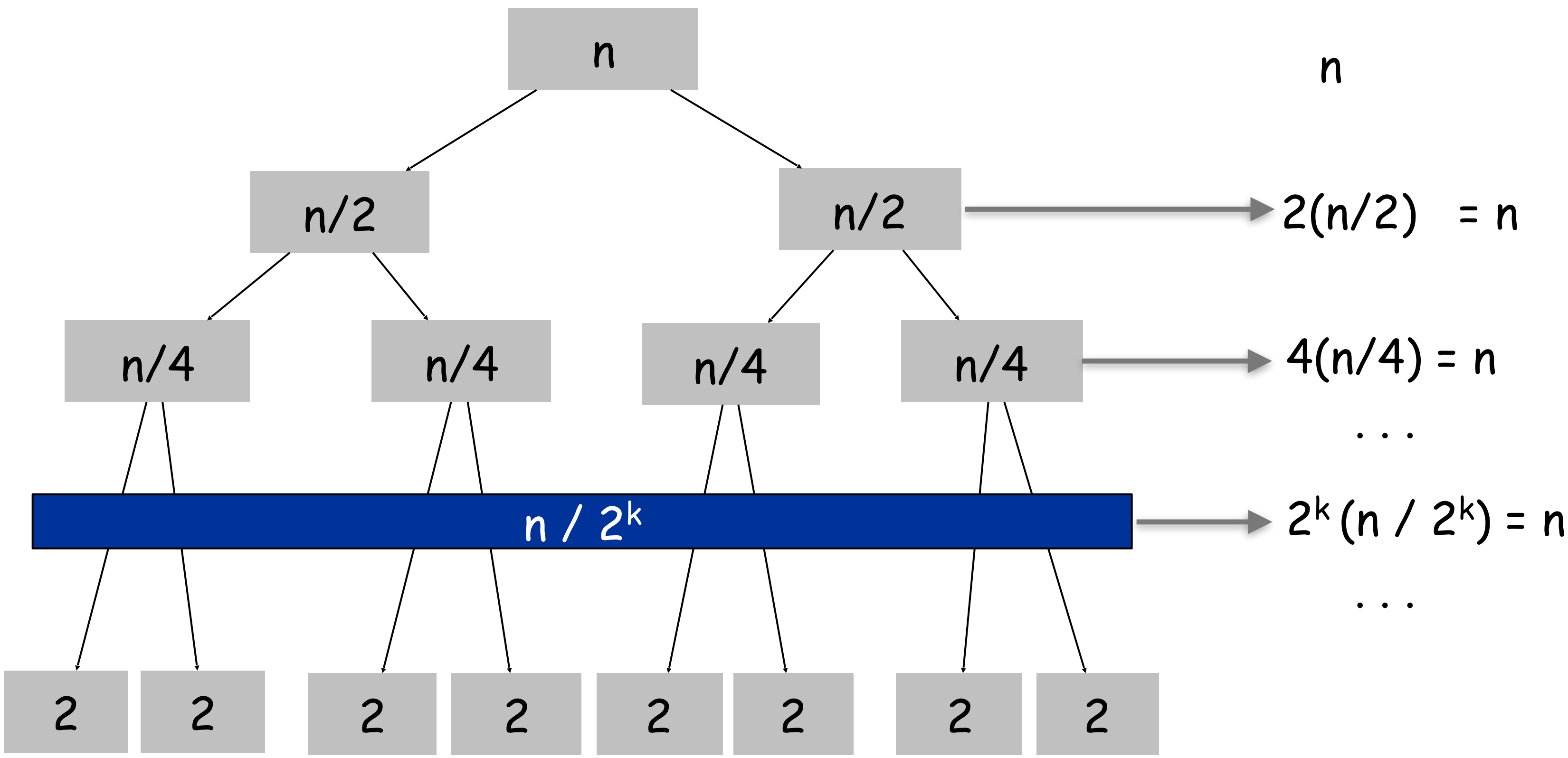
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



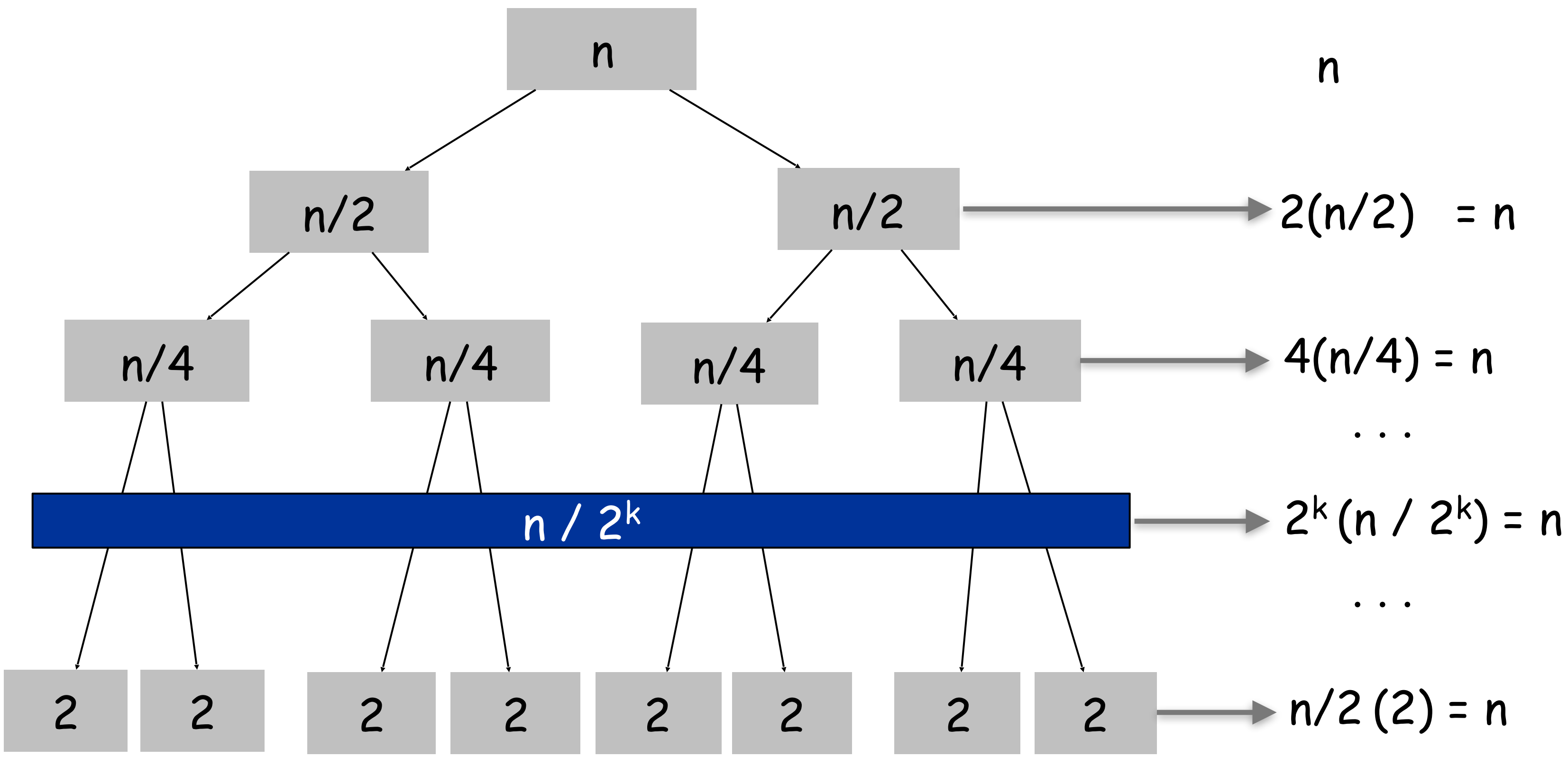
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



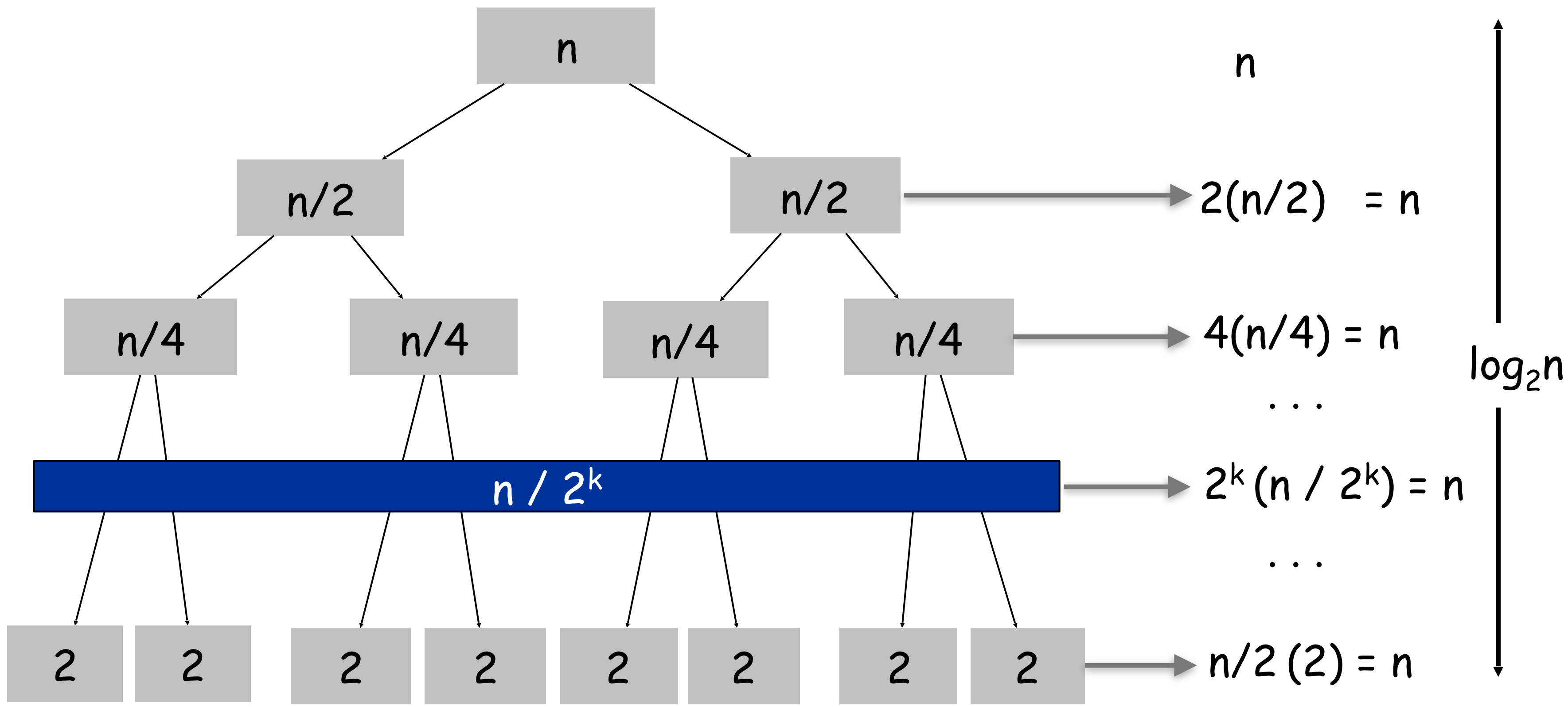
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



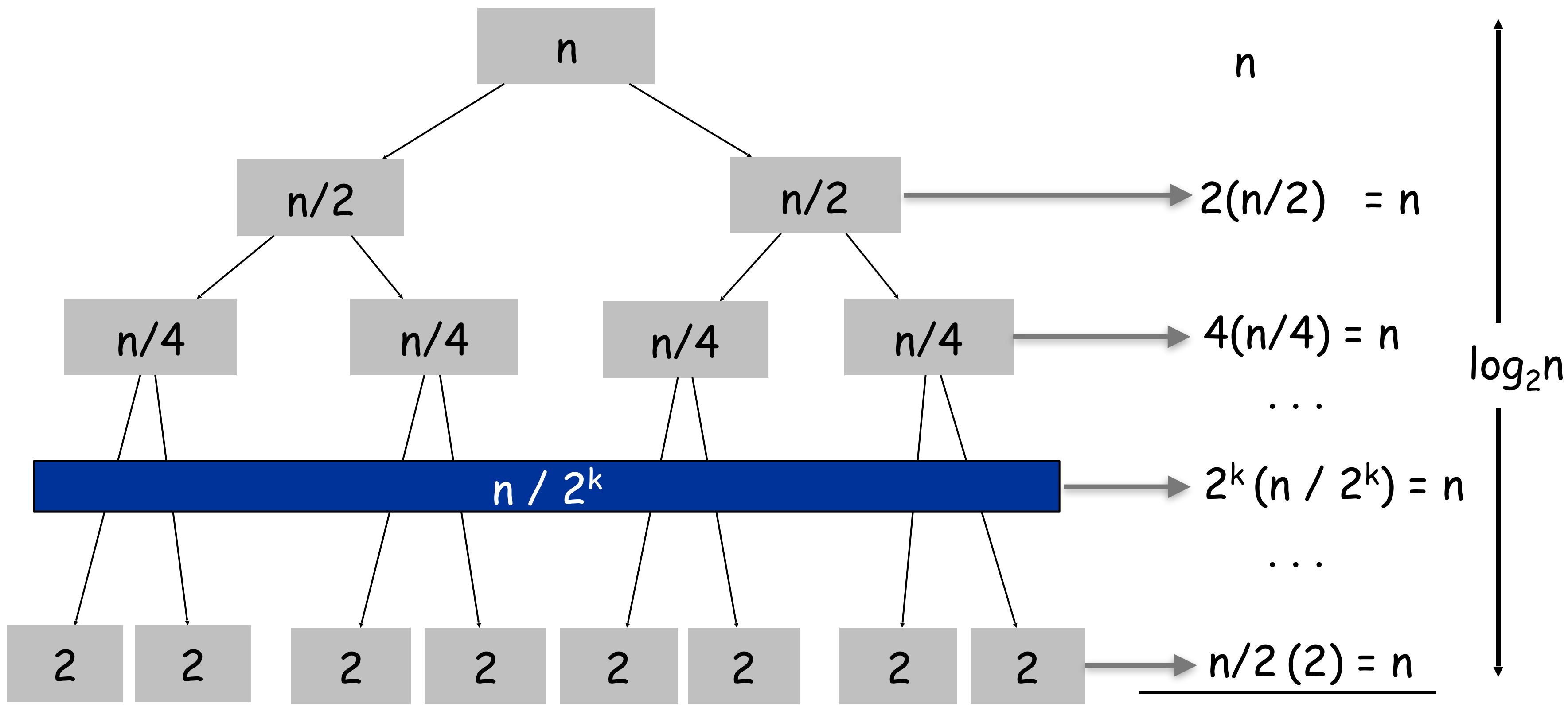
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



TOTAL = $n \log_2 n$

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

Terms.

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) \;+\; f(n)}$$

Terms.

- $a \geq 1$ is the number of subproblems.

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

Terms.

- $a \geq 1$ is the number of subproblems.
- $b > 0$ is the factor by which subproblem size decreases.

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

Terms.

- $a \geq 1$ is the number of subproblems.
- $b > 0$ is the factor by which subproblem size decreases.
- $f(n)$ = work to divide/merge subproblems.

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

Terms.

- $a \geq 1$ is the number of subproblems.
- $b > 0$ is the factor by which subproblem size decreases.
- $f(n)$ = work to divide/merge subproblems.

Recursion tree.

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

Terms.

- $a \geq 1$ is the number of subproblems.
- $b > 0$ is the factor by which subproblem size decreases.
- $f(n)$ = work to divide/merge subproblems.

Recursion tree.

- $\boldsymbol{t = \log_b n}$ levels.

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) \;+\; f(n)}$$

Terms.

- $a \geq 1$ is the number of subproblems.
- $b > 0$ is the factor by which subproblem size decreases.
- $f(n)$ = work to divide/merge subproblems.

Recursion tree.

- $t = \log_b n$ levels.
- a^i = number of subproblems at level i.

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

Terms.

- $a \geq 1$ is the number of subproblems.
- $b > 0$ is the factor by which subproblem size decreases.
- $f(n)$ = work to divide/merge subproblems.

Recursion tree.

- $t = \log_b n$ levels.
- a^i = number of subproblems at level i.
- n / b^i = size of subproblem at level i.

Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\ T(n / 2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.

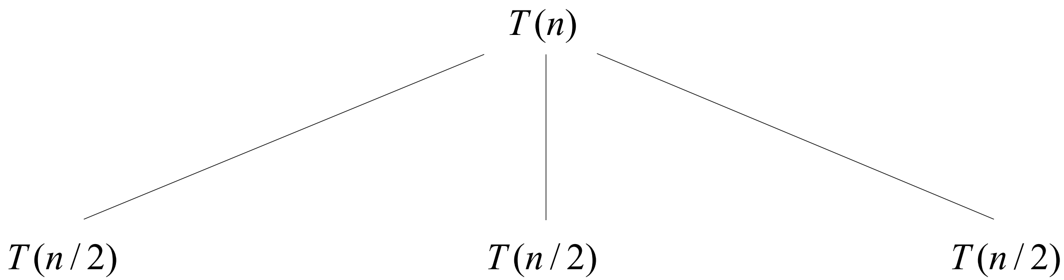
Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $\boldsymbol{T(n) = 3\,T(n/2) + n.}$ Then, $\boldsymbol{T(n) = \Theta(n^{\lg 3})}$.

$T(n)$

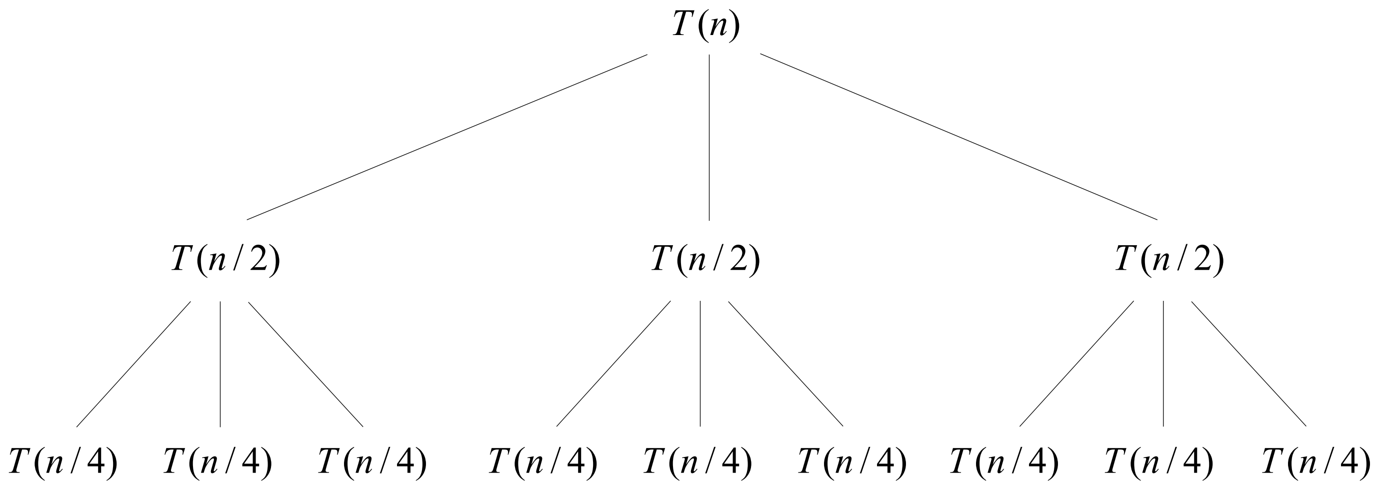
Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\ T(n / 2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



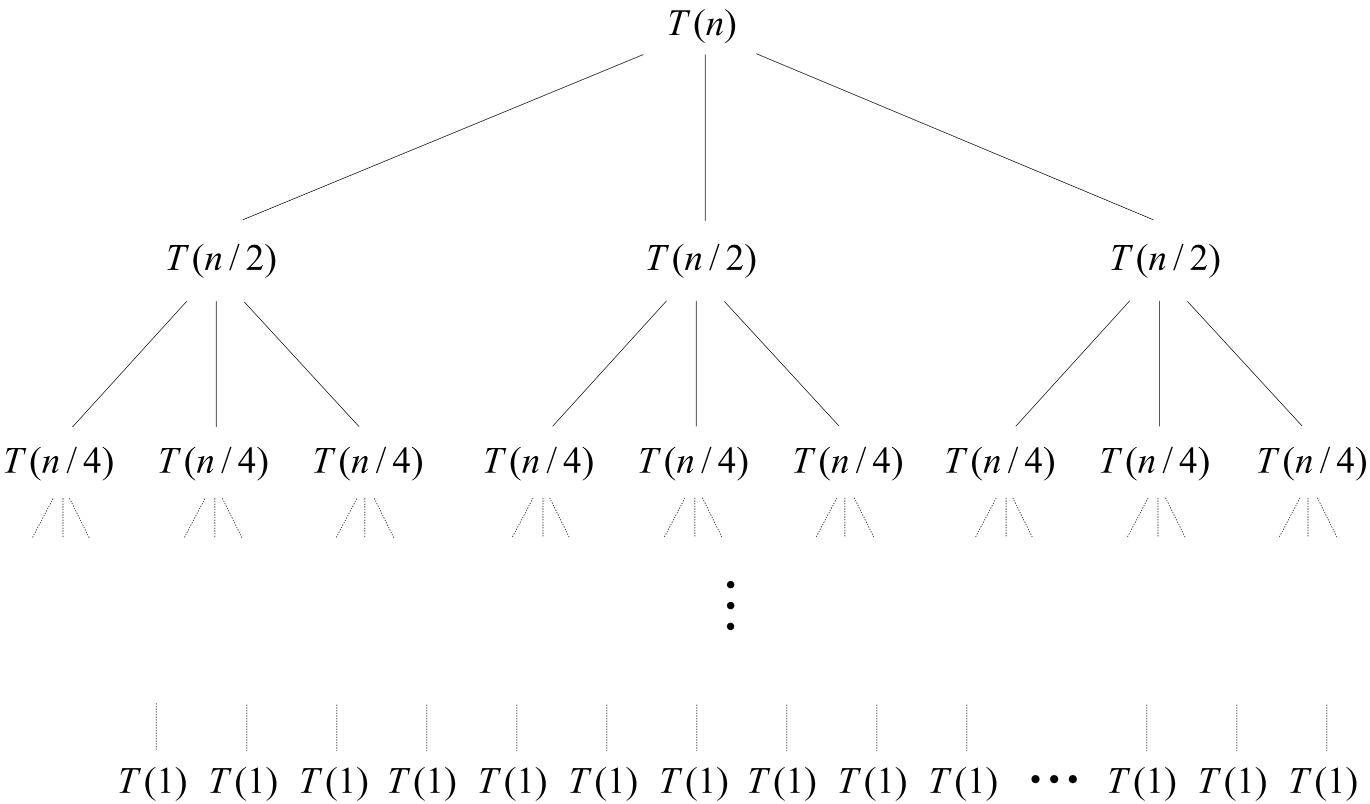
Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



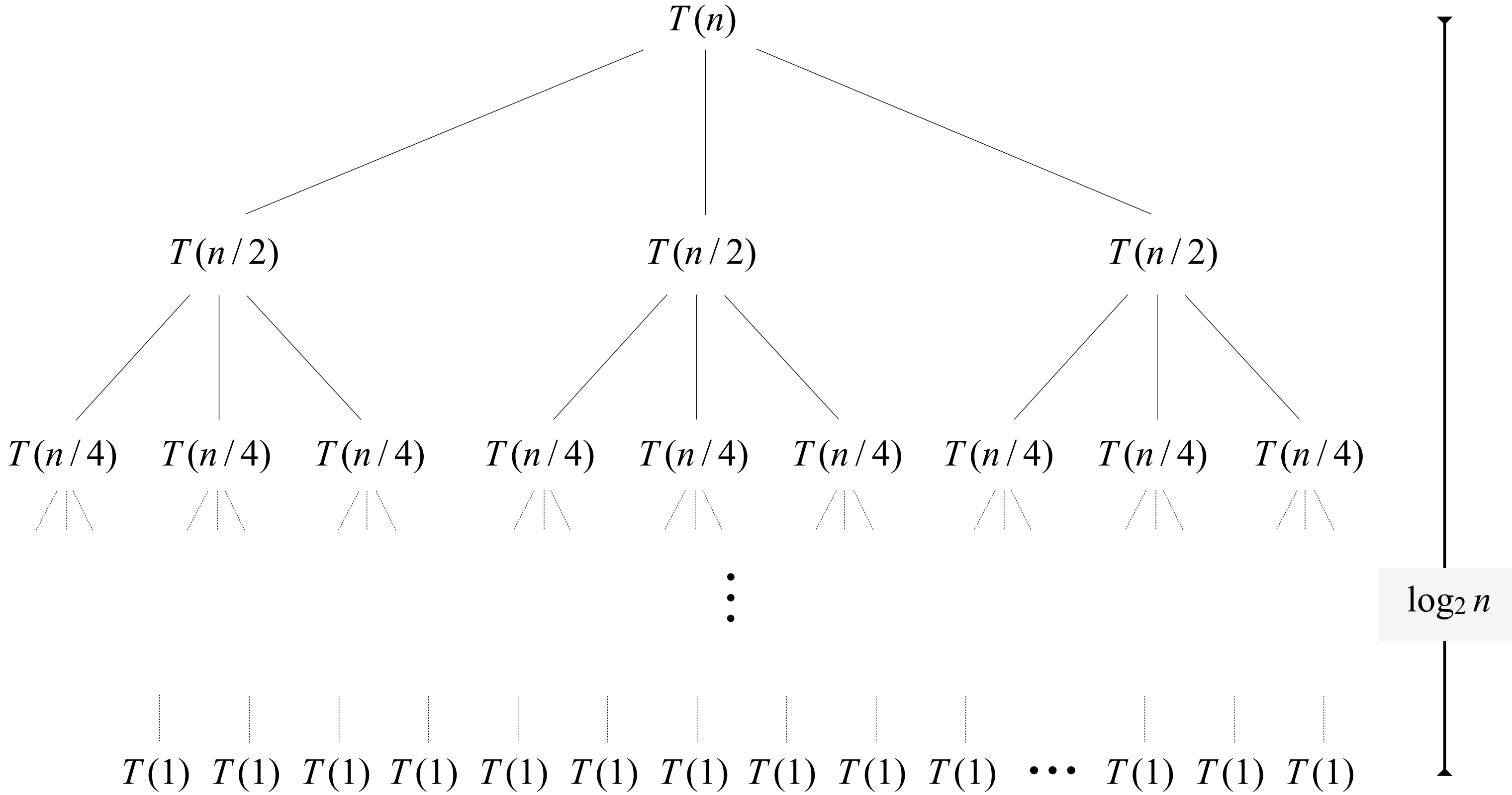
Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



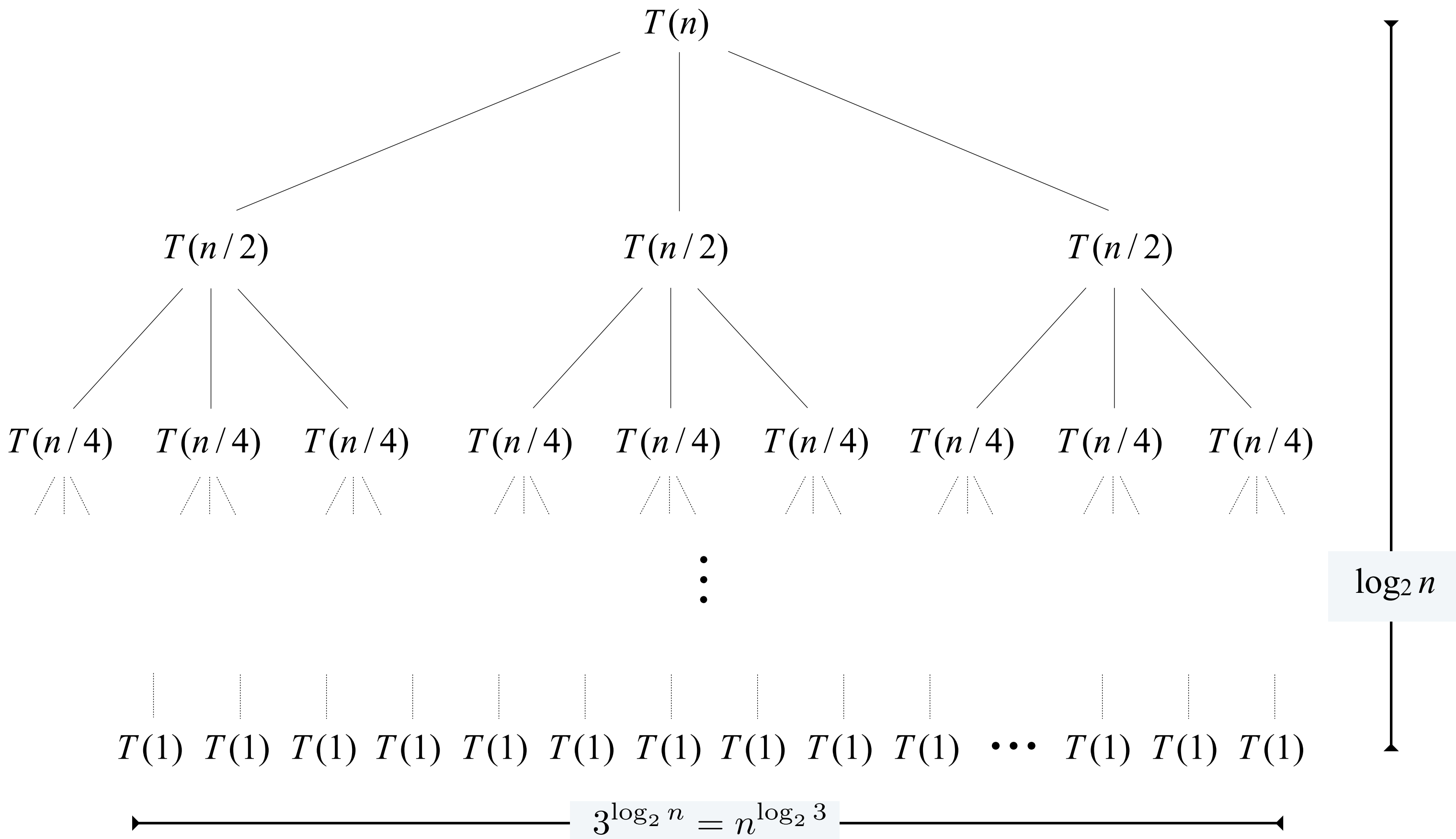
Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



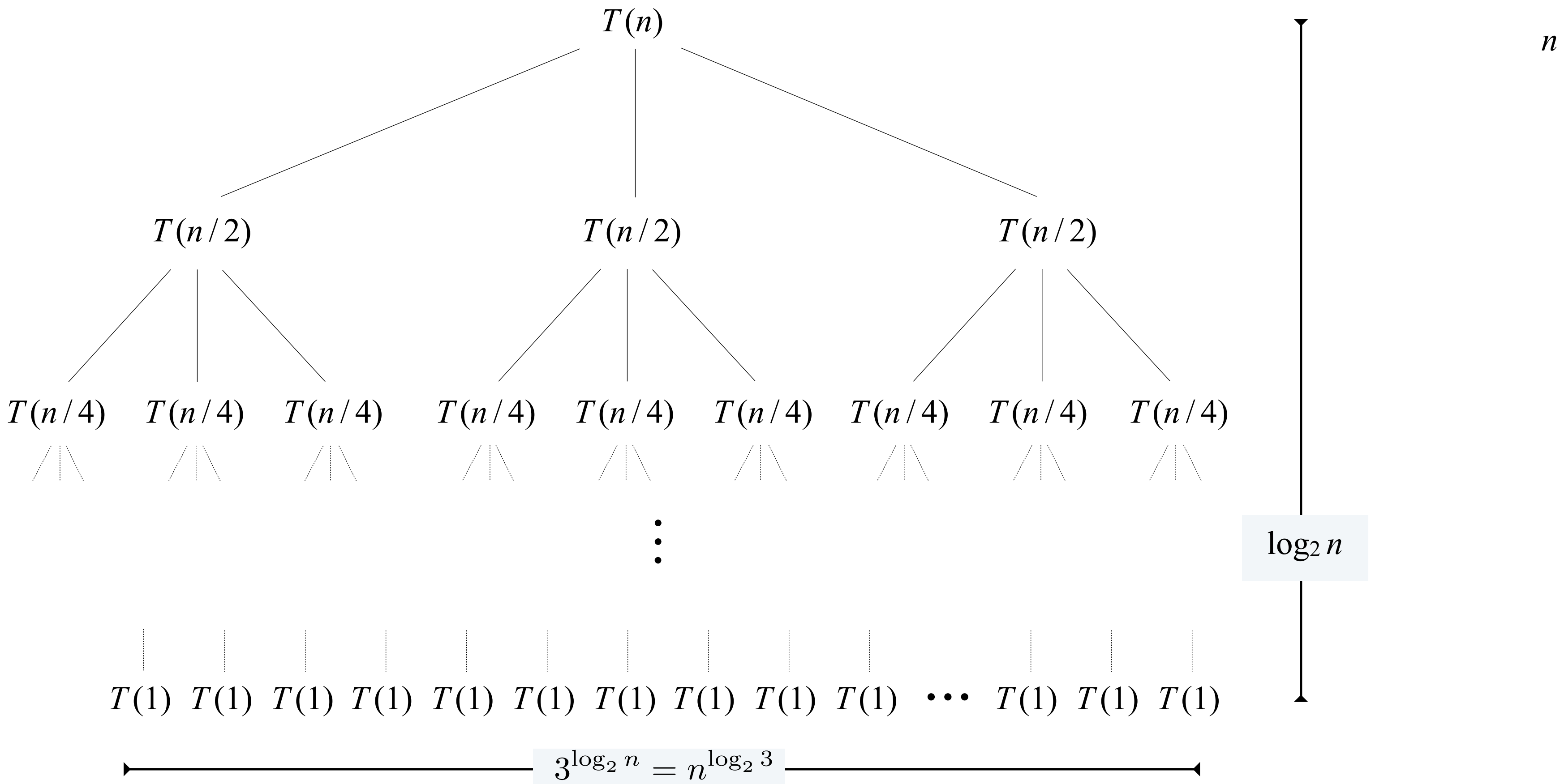
Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



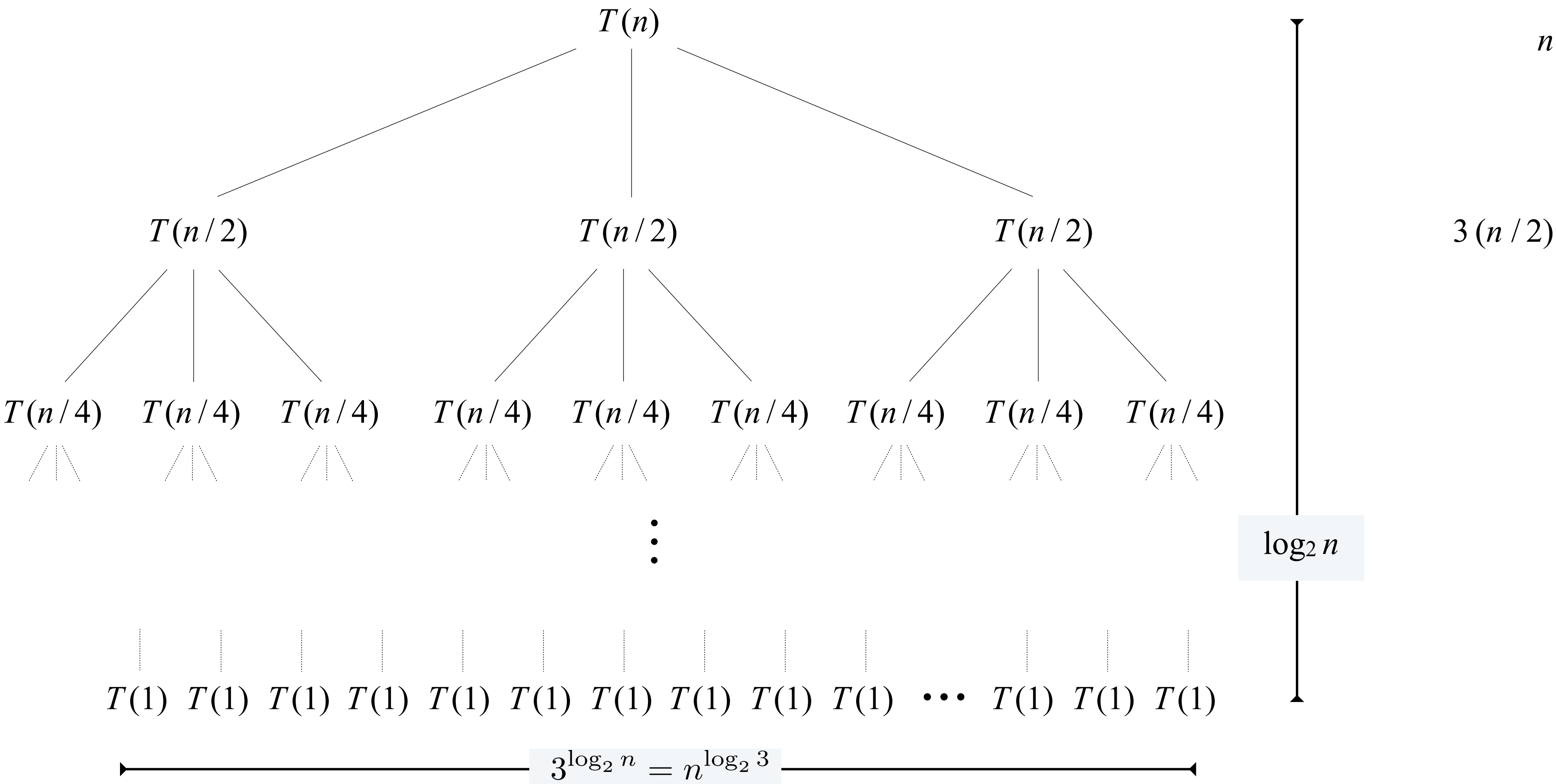
Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



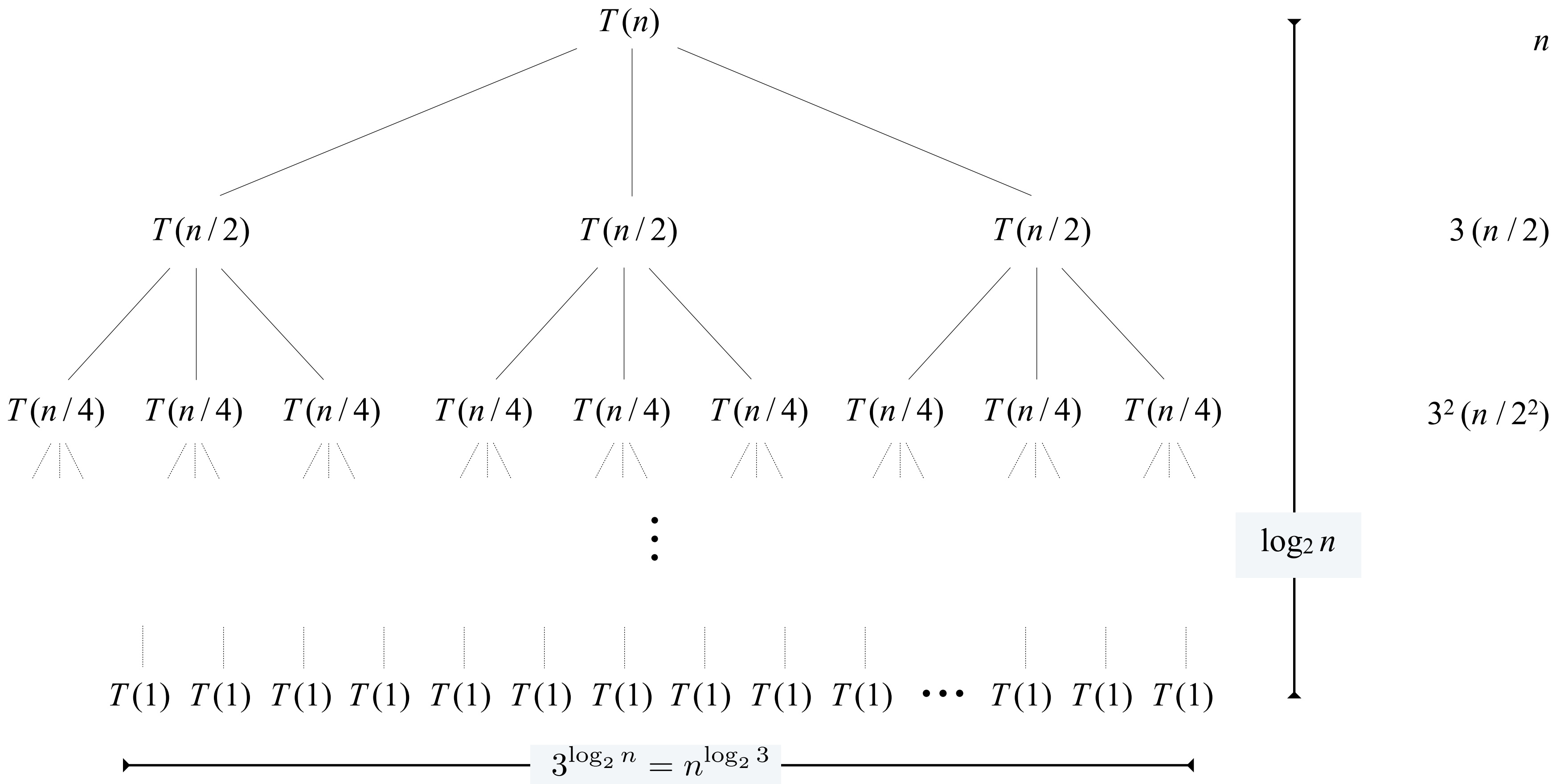
Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



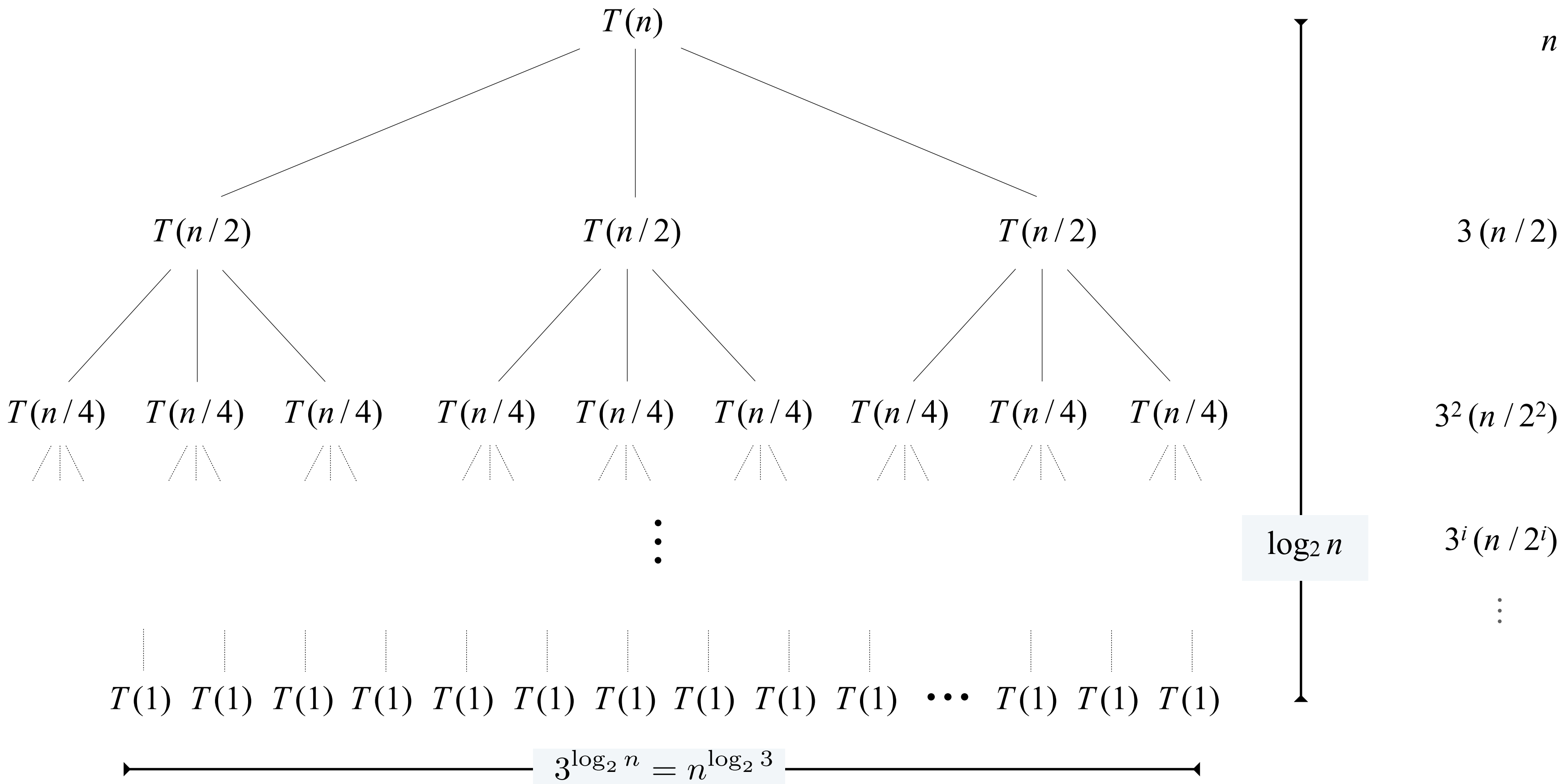
Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



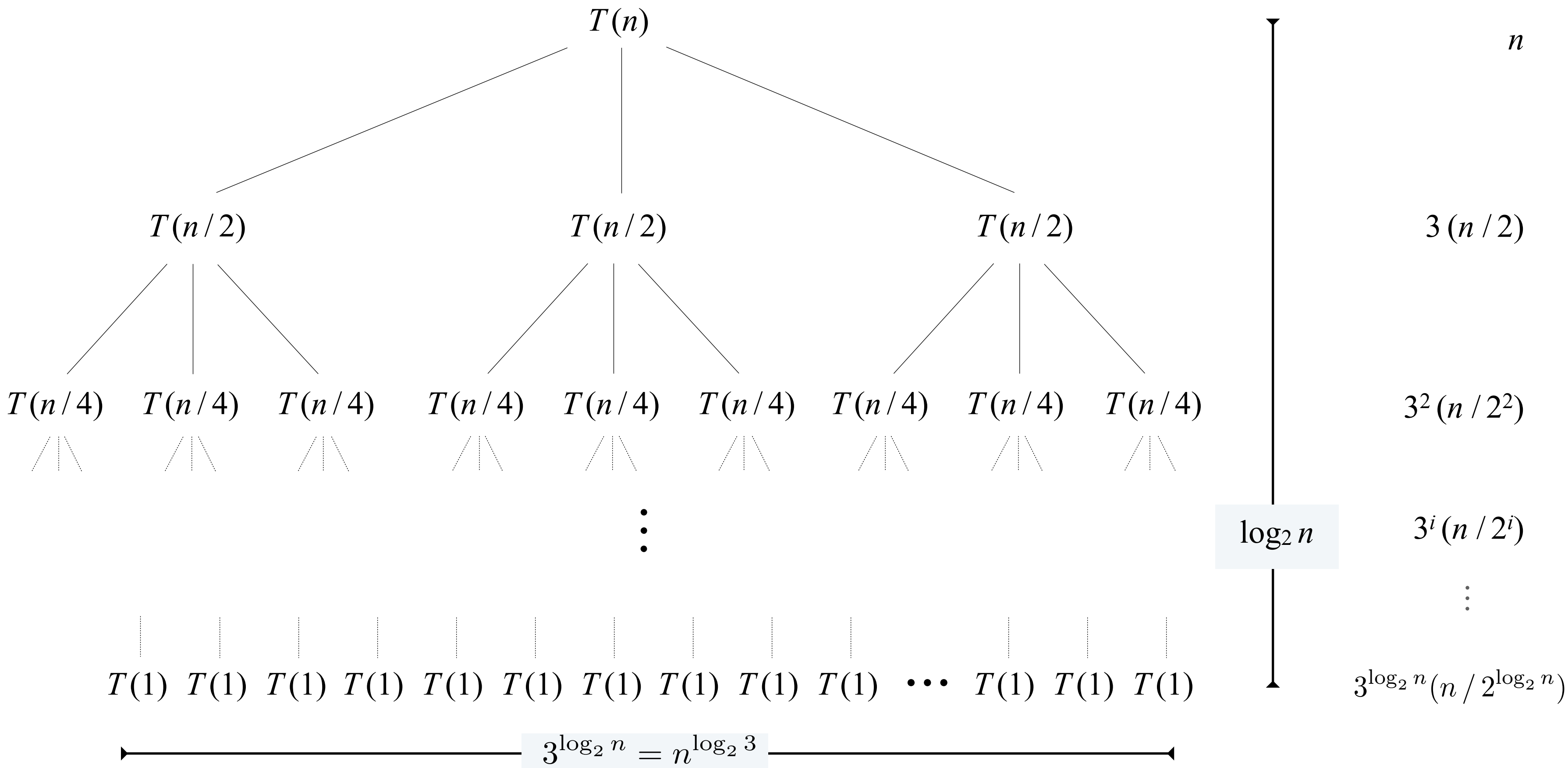
Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



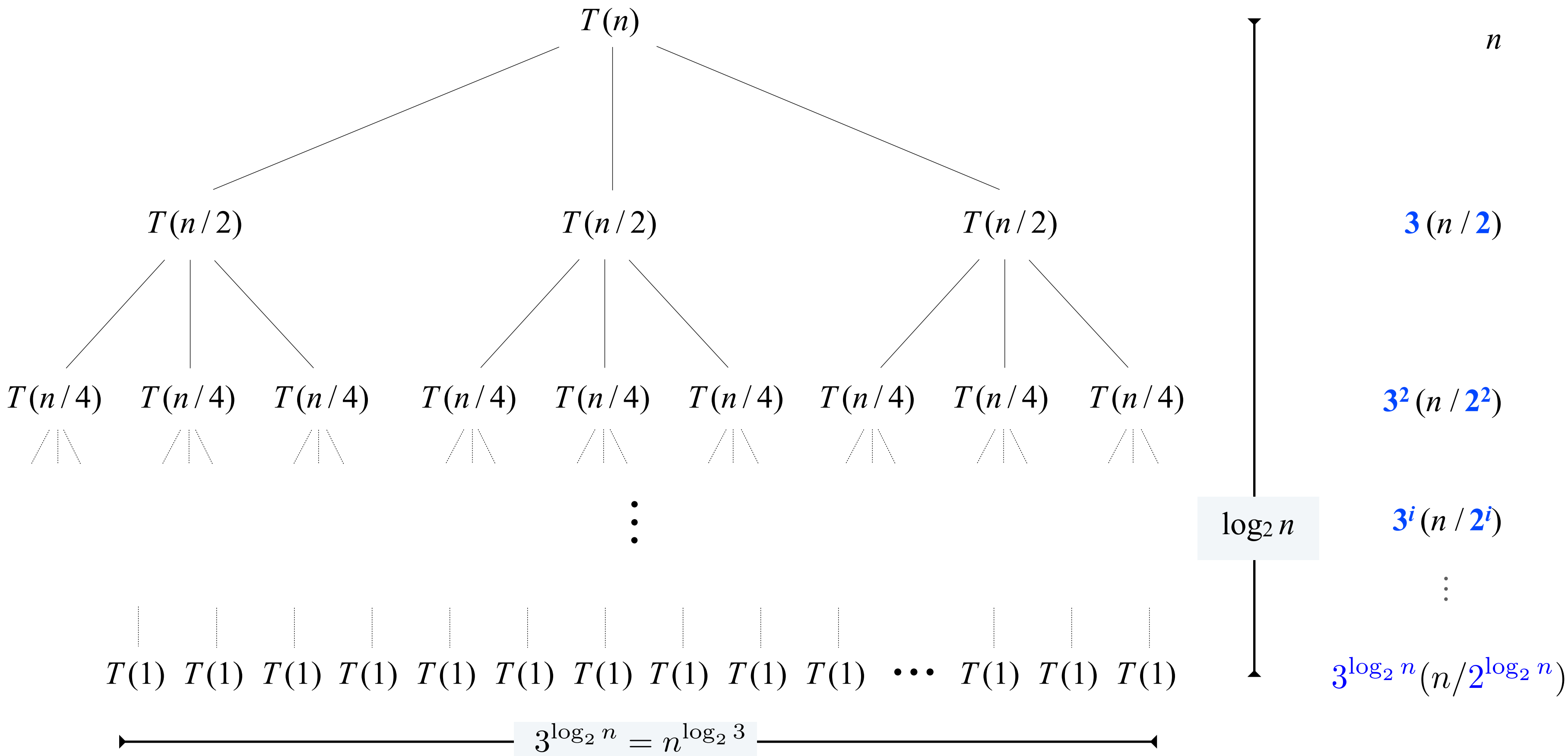
Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



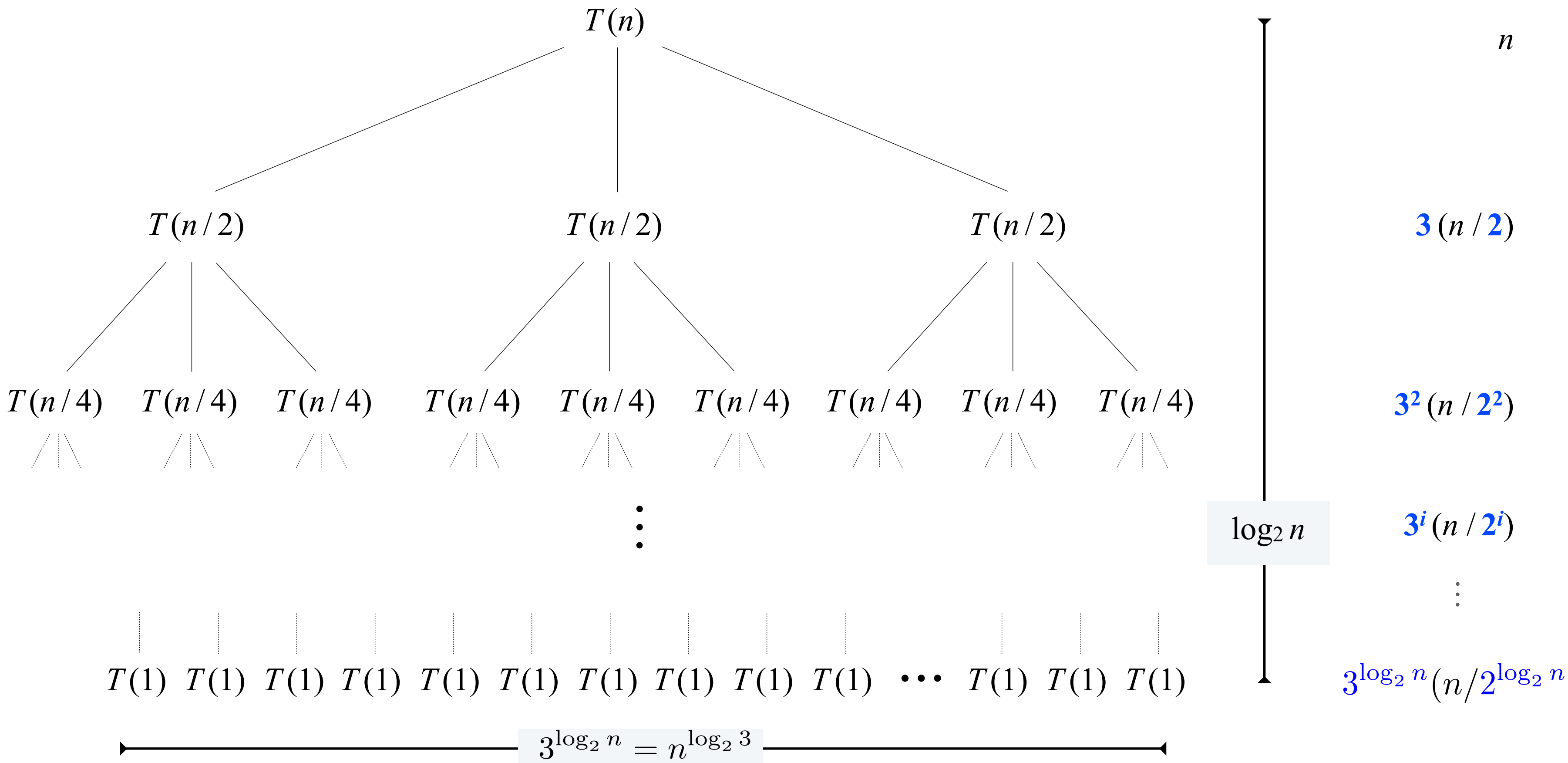
Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



Case 1: Cost of leaves dominates total cost

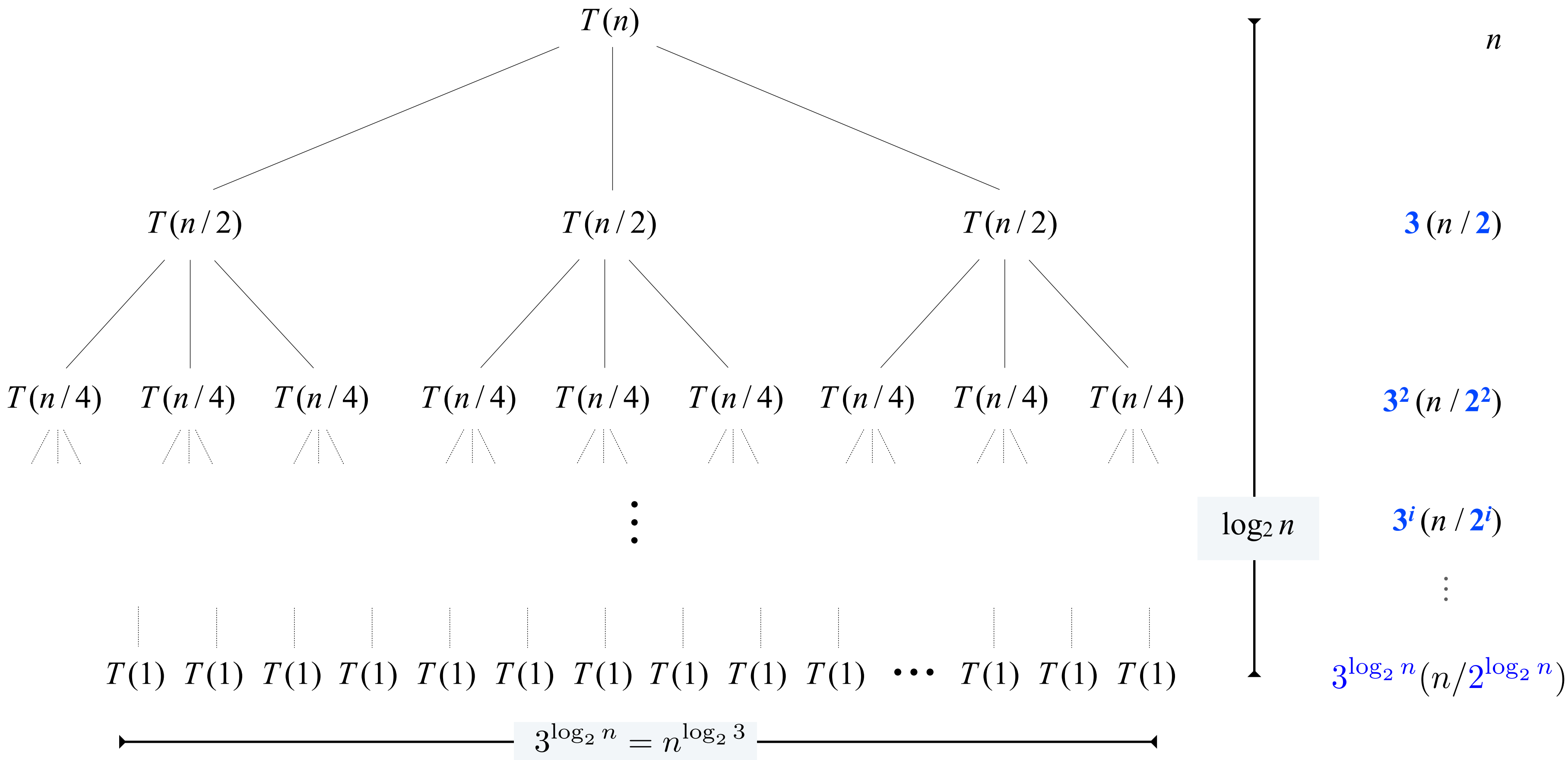
Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



$r = 3/2 > 1$

Case 1: Cost of leaves dominates total cost

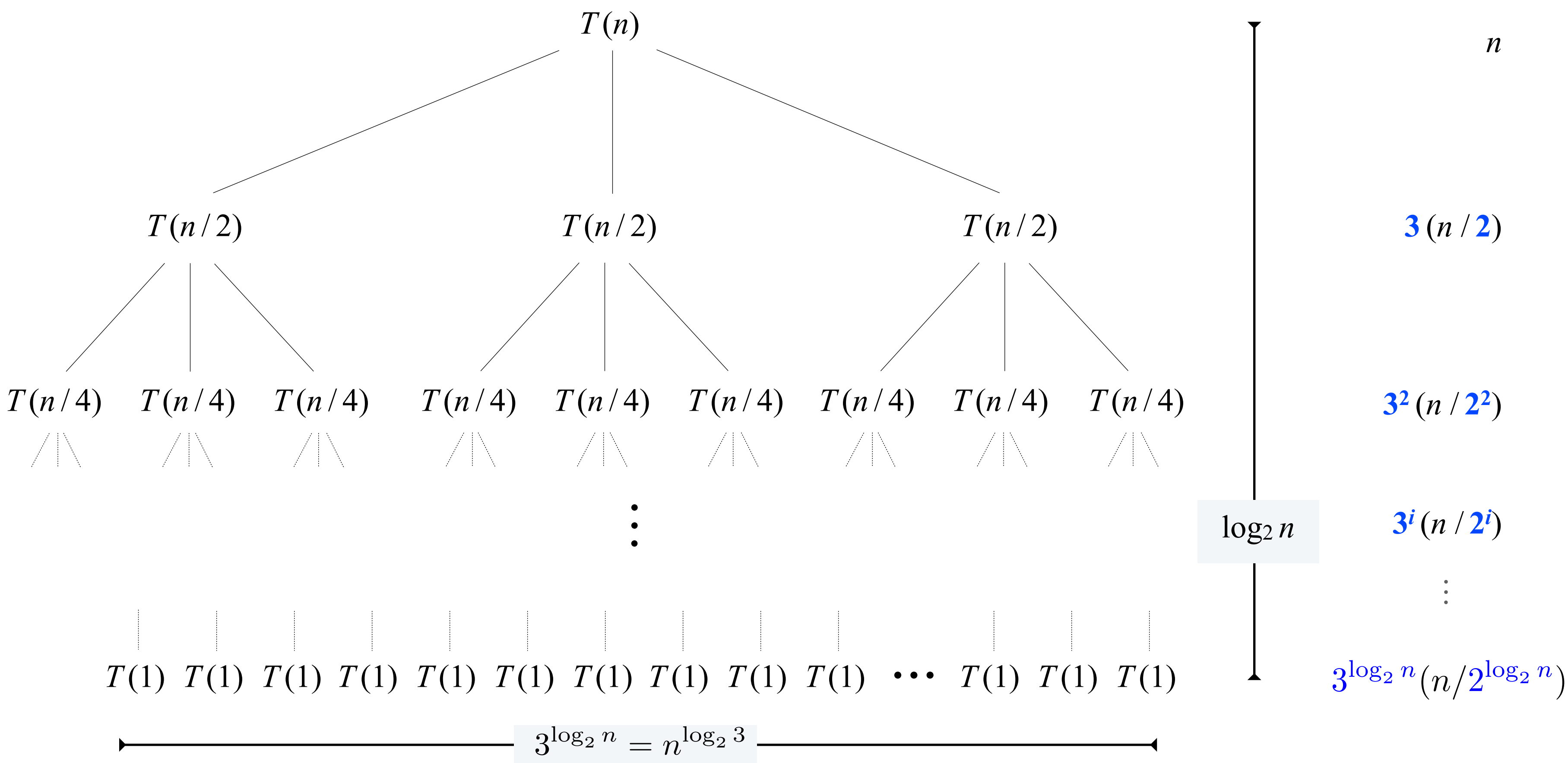
Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



$r = 3/2 > 1$ $\quad T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\, n$

Case 1: Cost of leaves dominates total cost

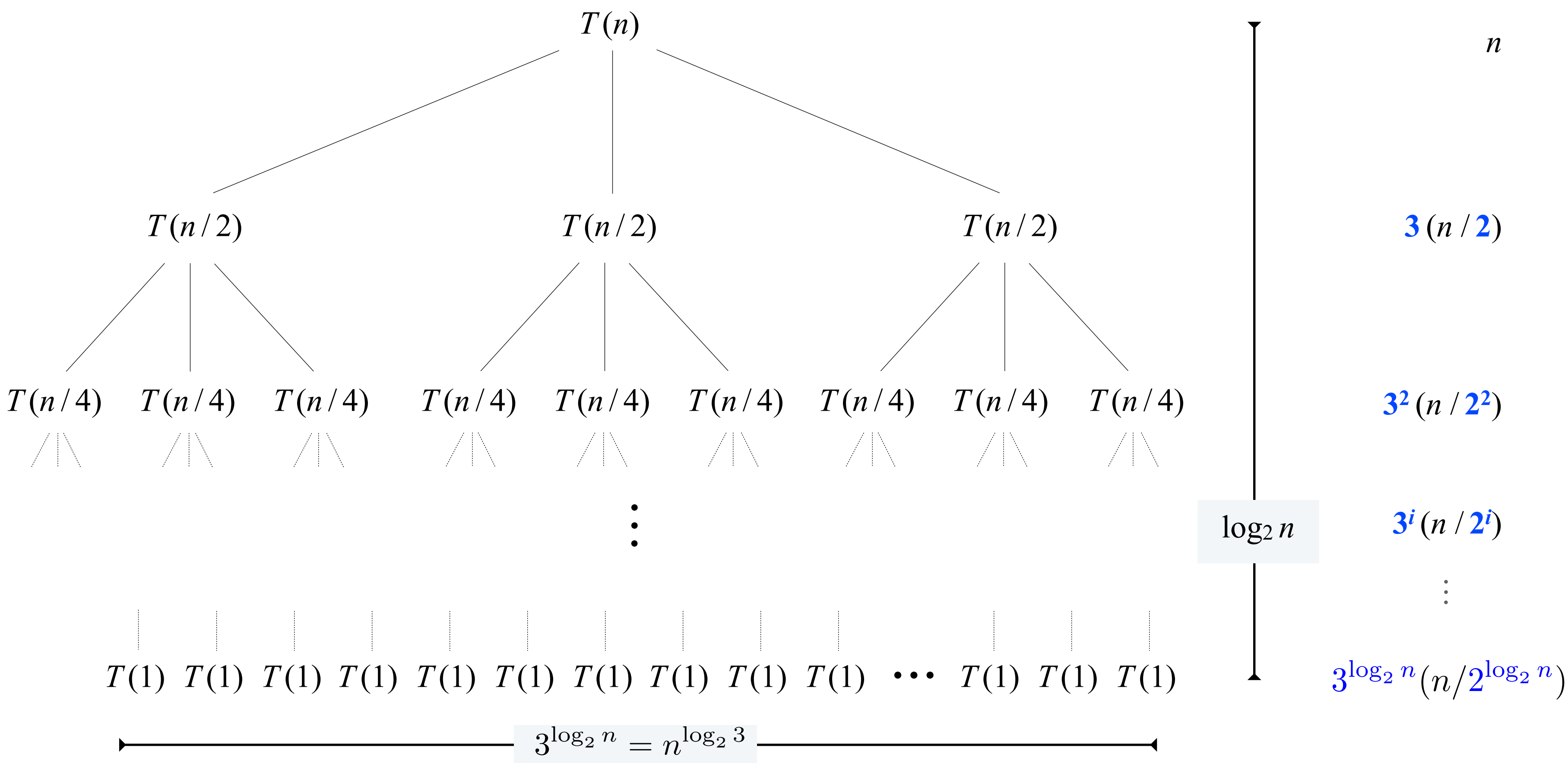
Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



$r = 3/2 > 1$ $\quad T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\, n = \dfrac{r^{1+\log_2 n} - 1}{r - 1}\, n$

Case 1: Cost of leaves dominates total cost

Example: T(1) = 1. $T(n) = 3\,T(n/2) + n$. Then, $T(n) = \Theta(n^{\lg 3})$.



$r = 3/2 > 1$ $\quad T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\,n = \dfrac{r^{1+\log_2 n} - 1}{r - 1}\,n = 3n^{\log_2 3} - 2n$

Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $T(n) = 2\,T(n/2) + n$. Then $T(n) = \Theta(n \log n)$.

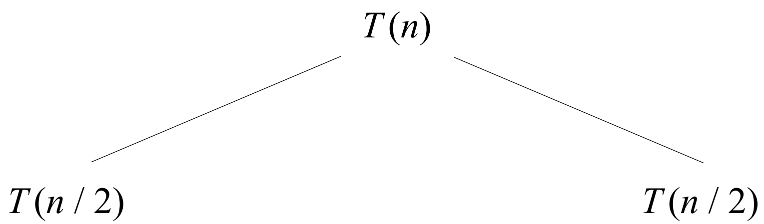
Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $T(n) = 2\,T(n/2) + n$. Then $T(n) = \Theta(n \log n)$.

$T(n)$

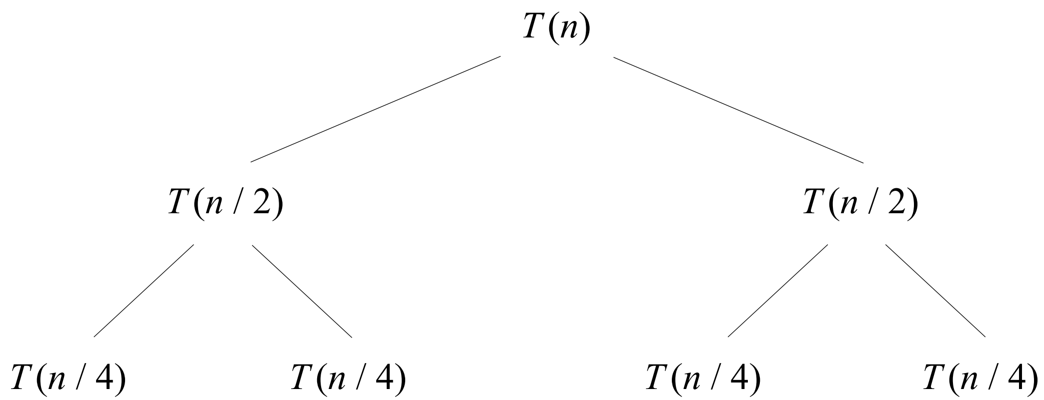
Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $T(n) = 2\,T(n/2) + n$. Then $T(n) = \Theta(n \log n)$.



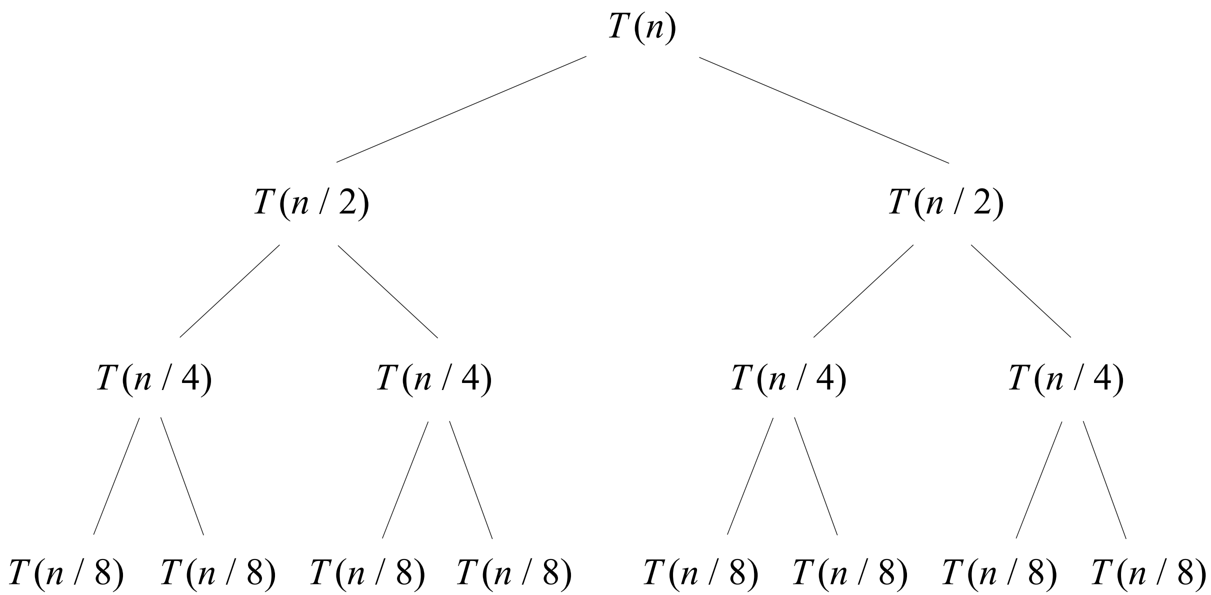
Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $T(n) = 2\,T(n/2) + n$. Then $T(n) = \Theta(n \log n)$.



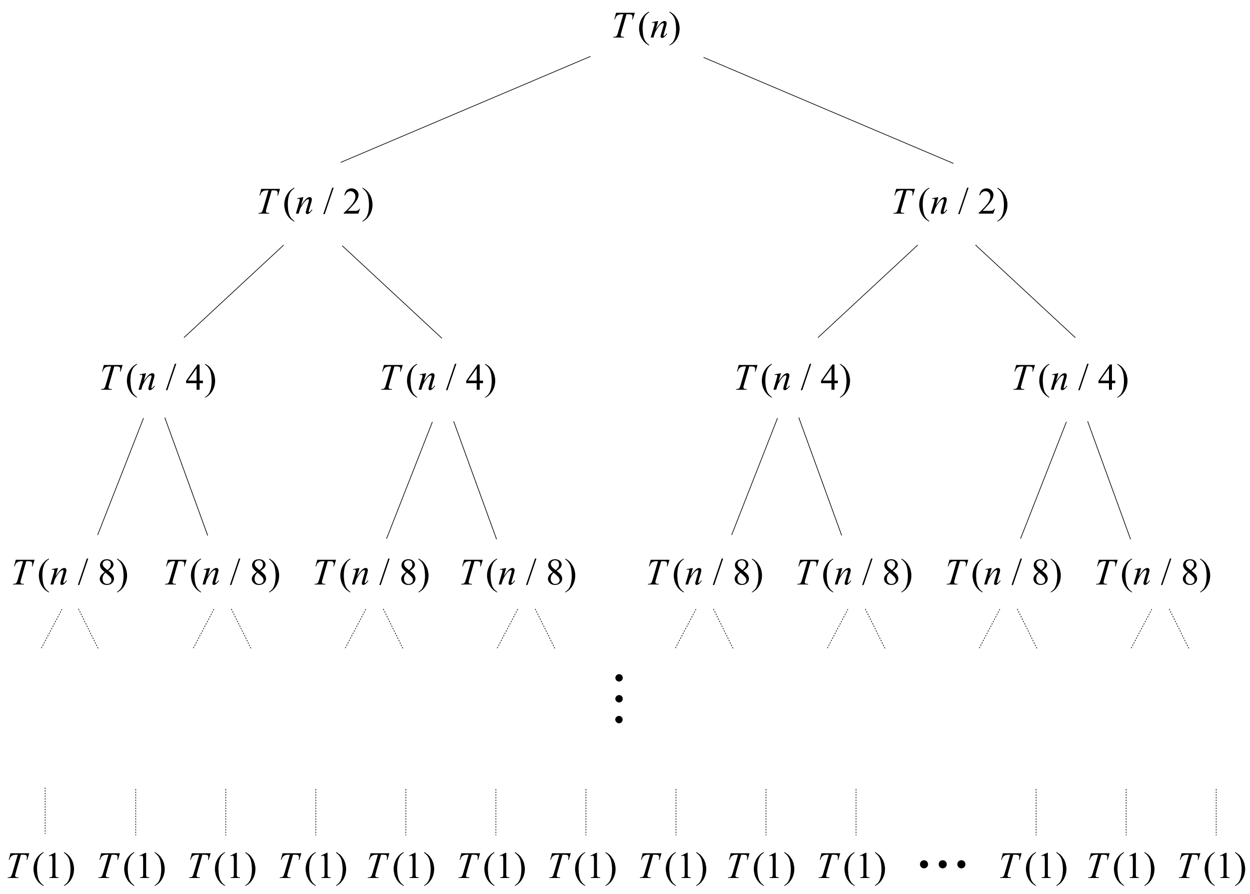
Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $\boldsymbol{T(n) = 2\,T(n/2) + n}$. Then $\boldsymbol{T(n) = \Theta(n \log n)}$.



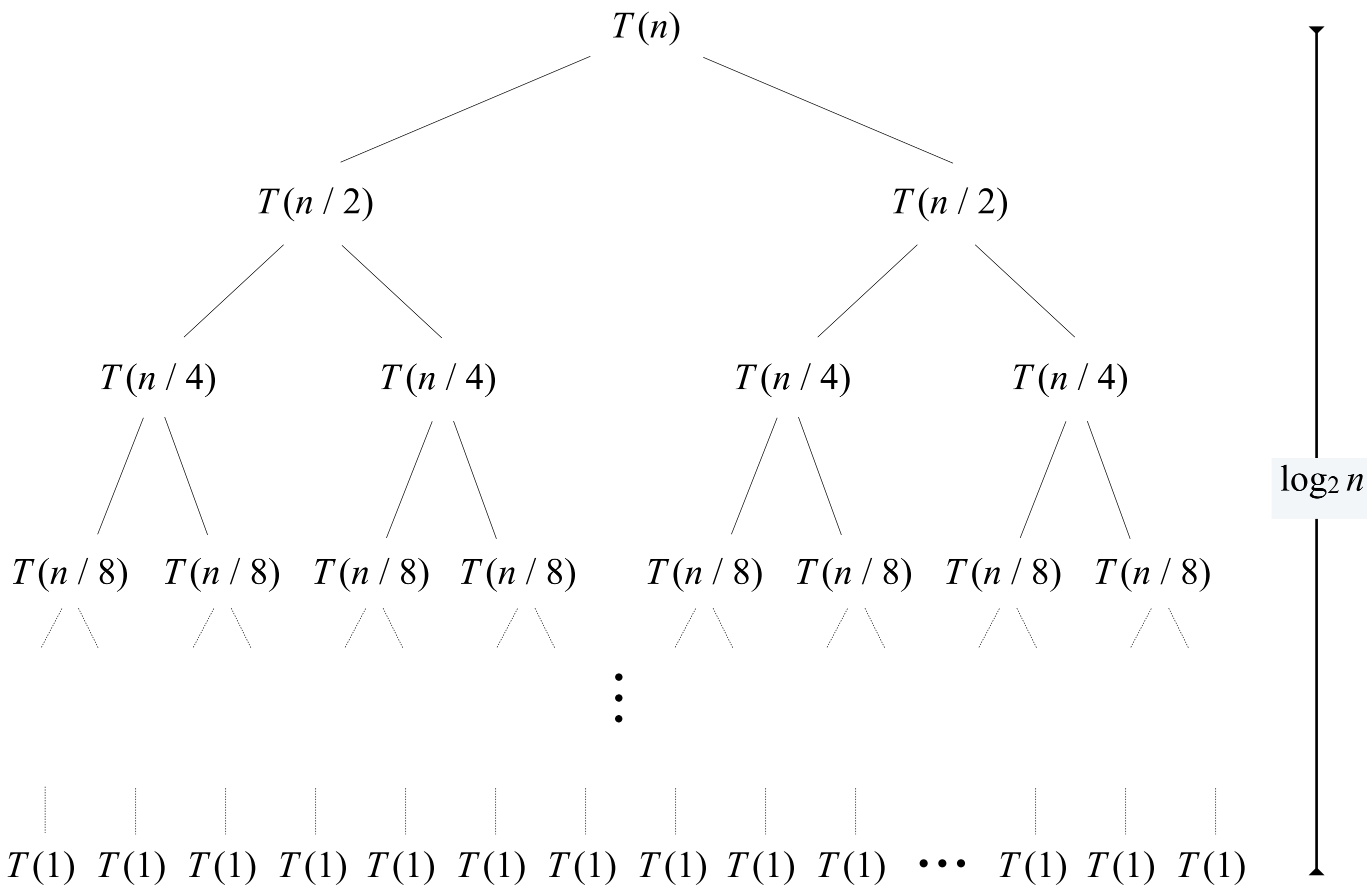
Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $\boldsymbol{T(n) = 2\,T(n/2) + n}$. Then $\boldsymbol{T(n) = \Theta(n \log n)}$.



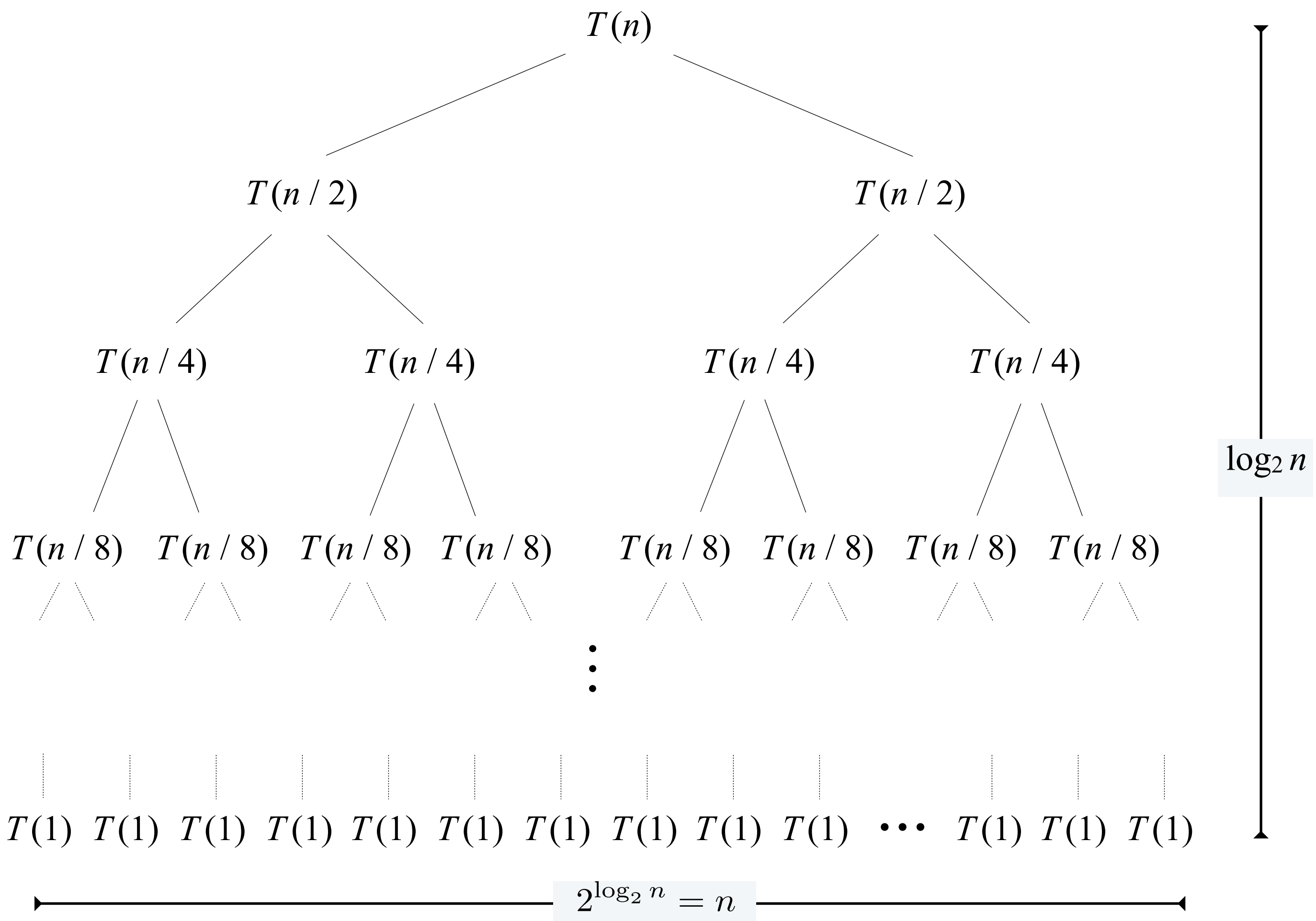
Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $T(n) = 2\,T(n/2) + n$. Then $T(n) = \Theta(n \log n)$.



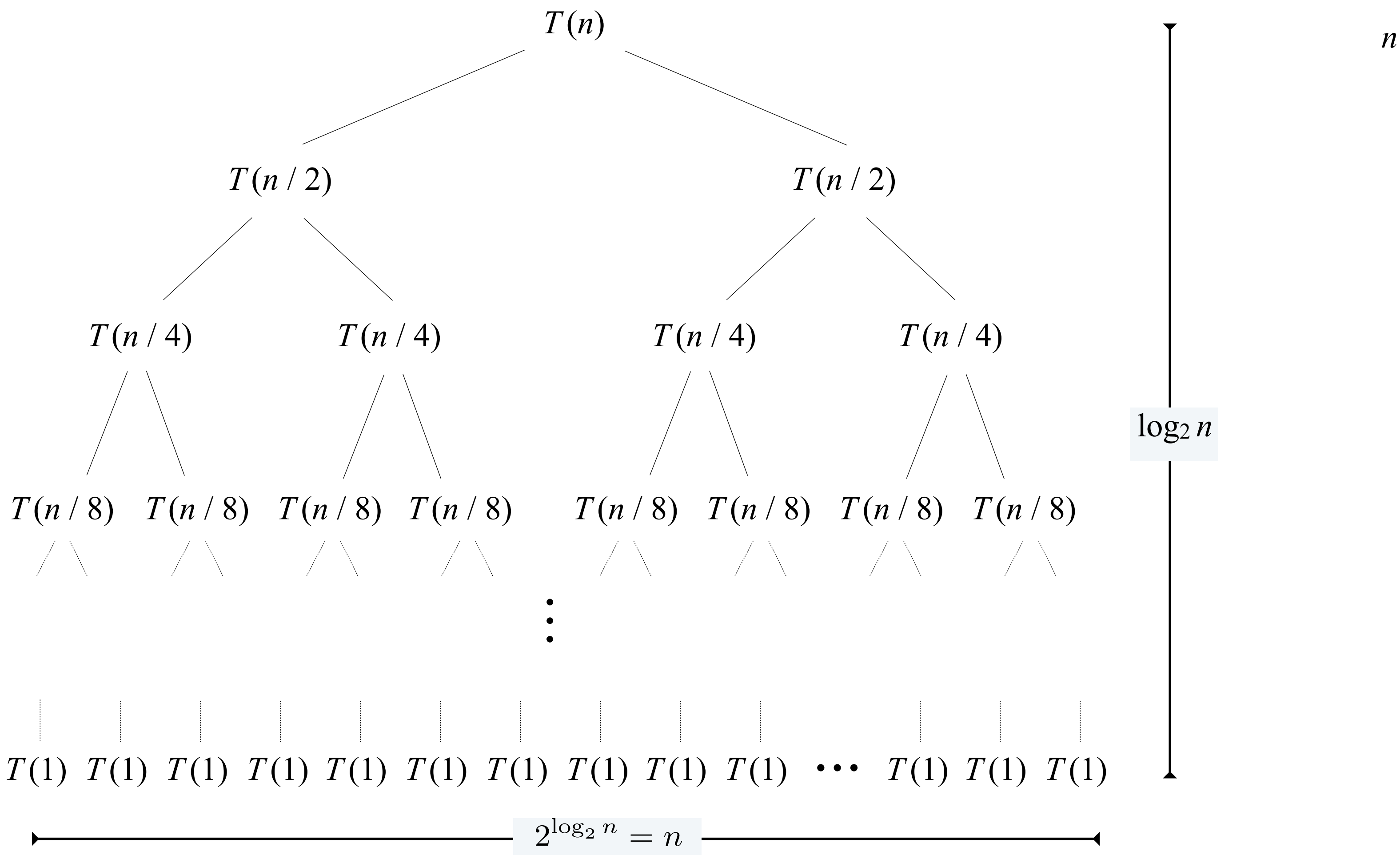
Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $T(n) = 2\,T(n/2) + n$. Then $T(n) = \Theta(n \log n)$.



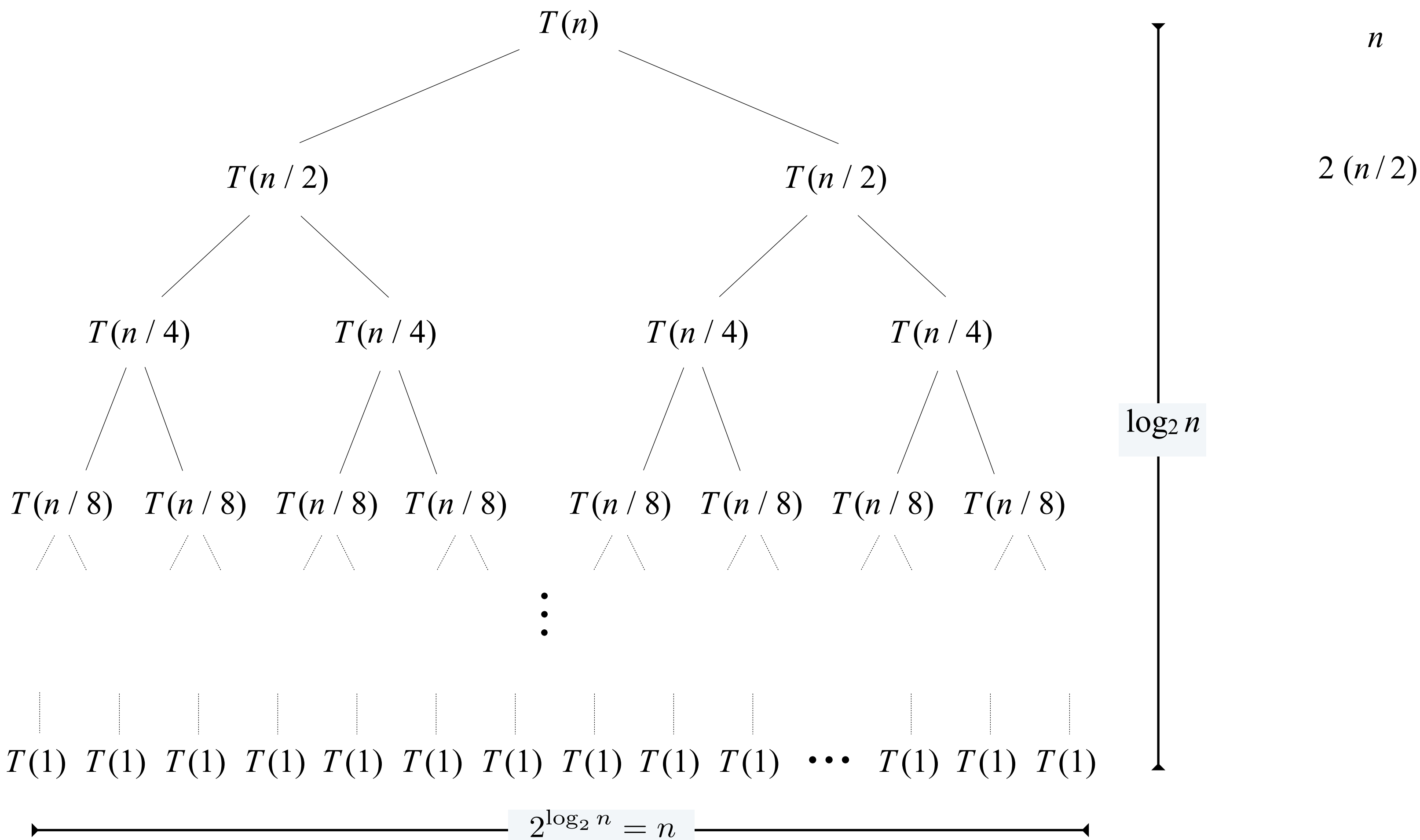
Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $\boldsymbol{T(n) = 2\,T(n/2) + n}$. Then $\boldsymbol{T(n) = \Theta(n \log n)}$.



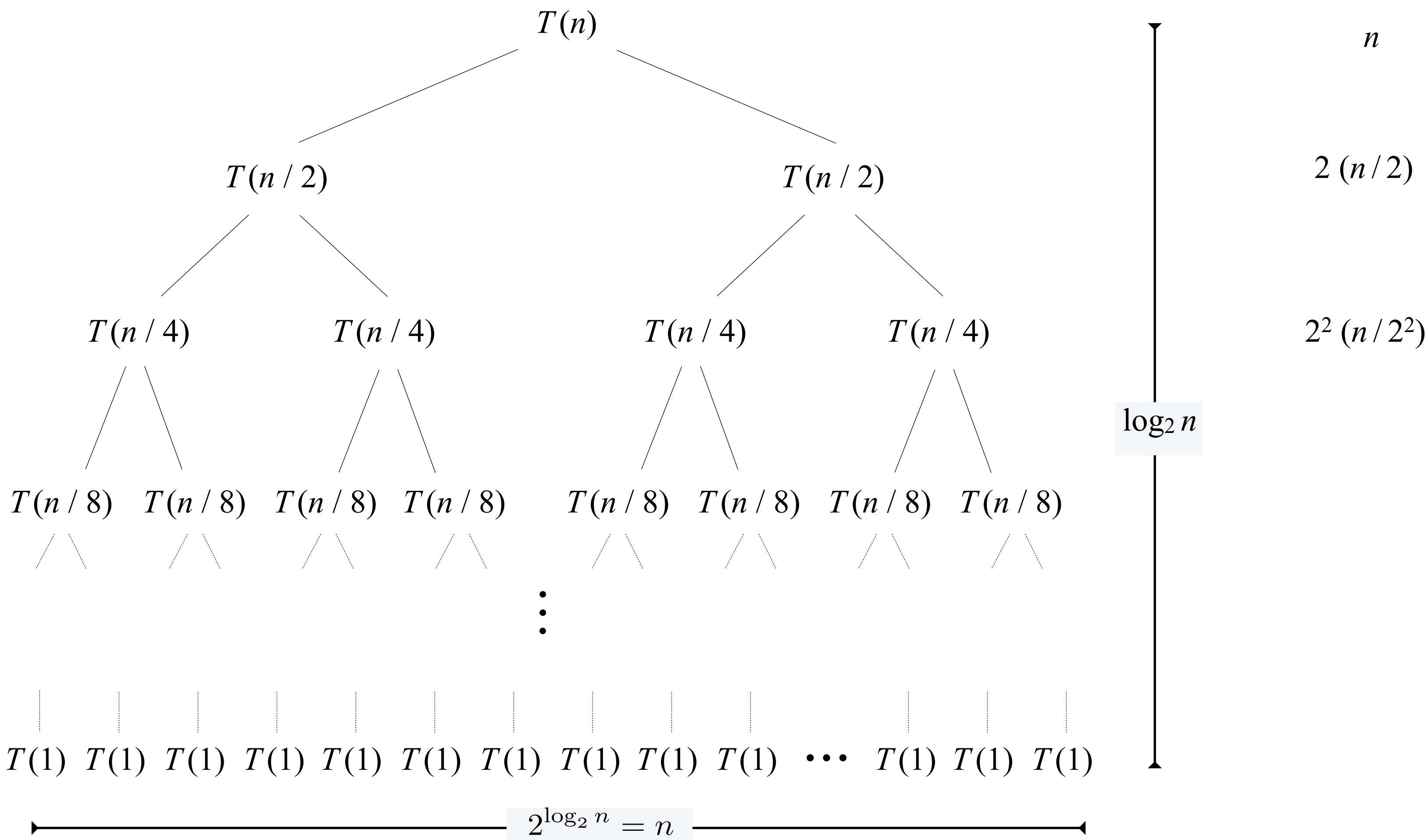
Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $T(n) = 2\,T(n/2) + n$. Then $T(n) = \Theta(n \log n)$.



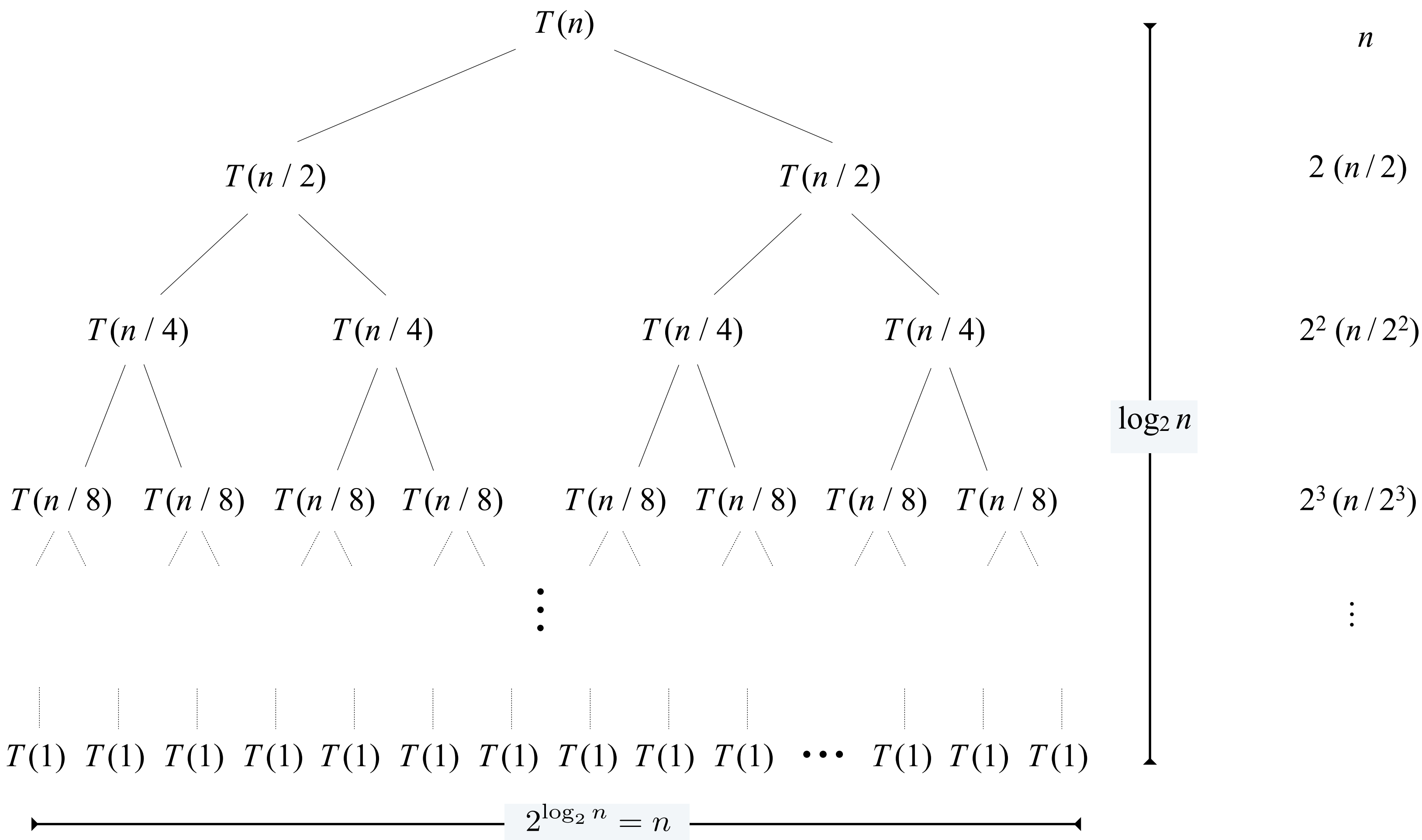
Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $T(n) = 2\,T(n/2) + n$. Then $T(n) = \Theta(n \log n)$.



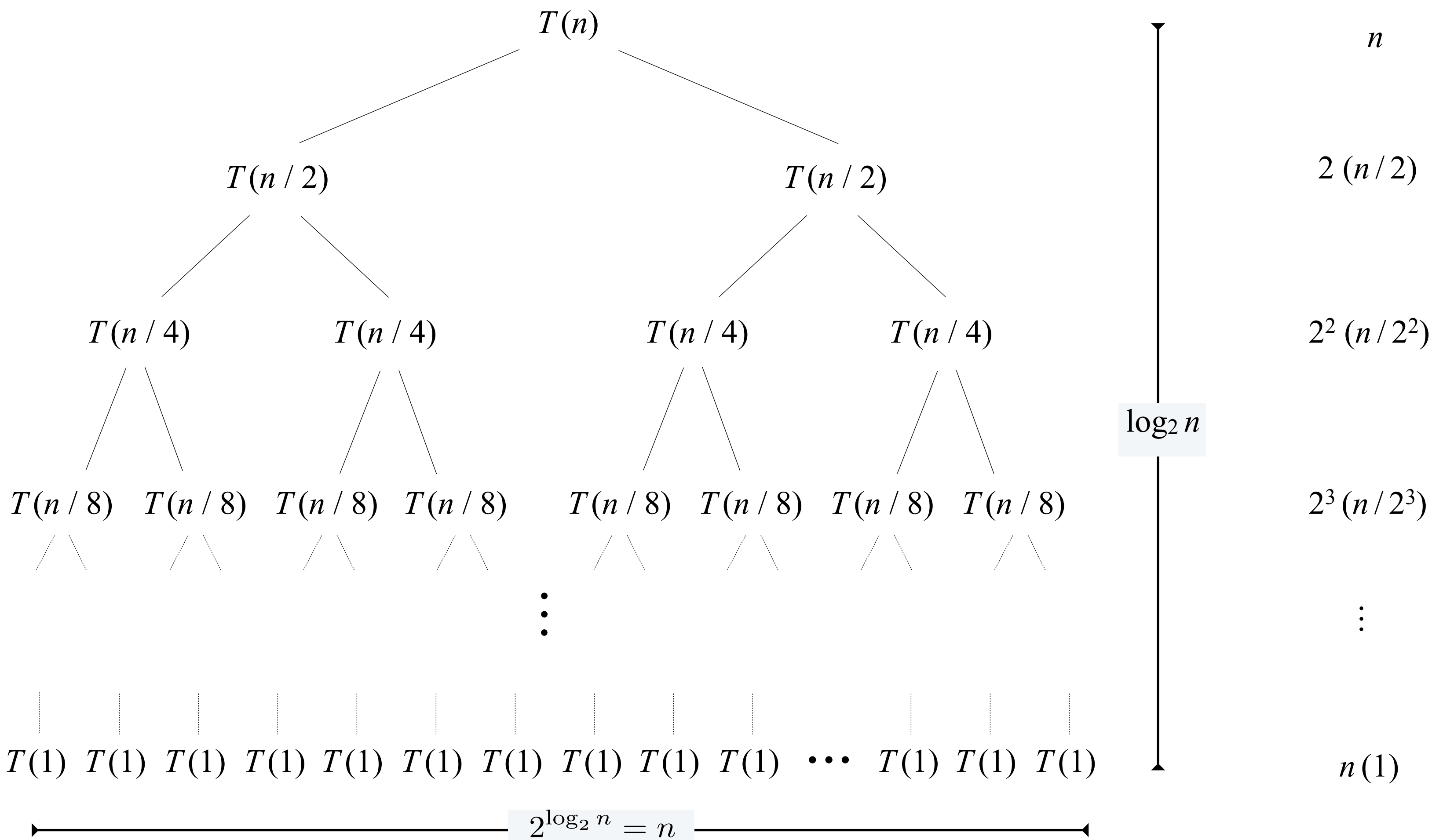
Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $T(n) = 2\,T(n/2) + n$. Then $T(n) = \Theta(n \log n)$.



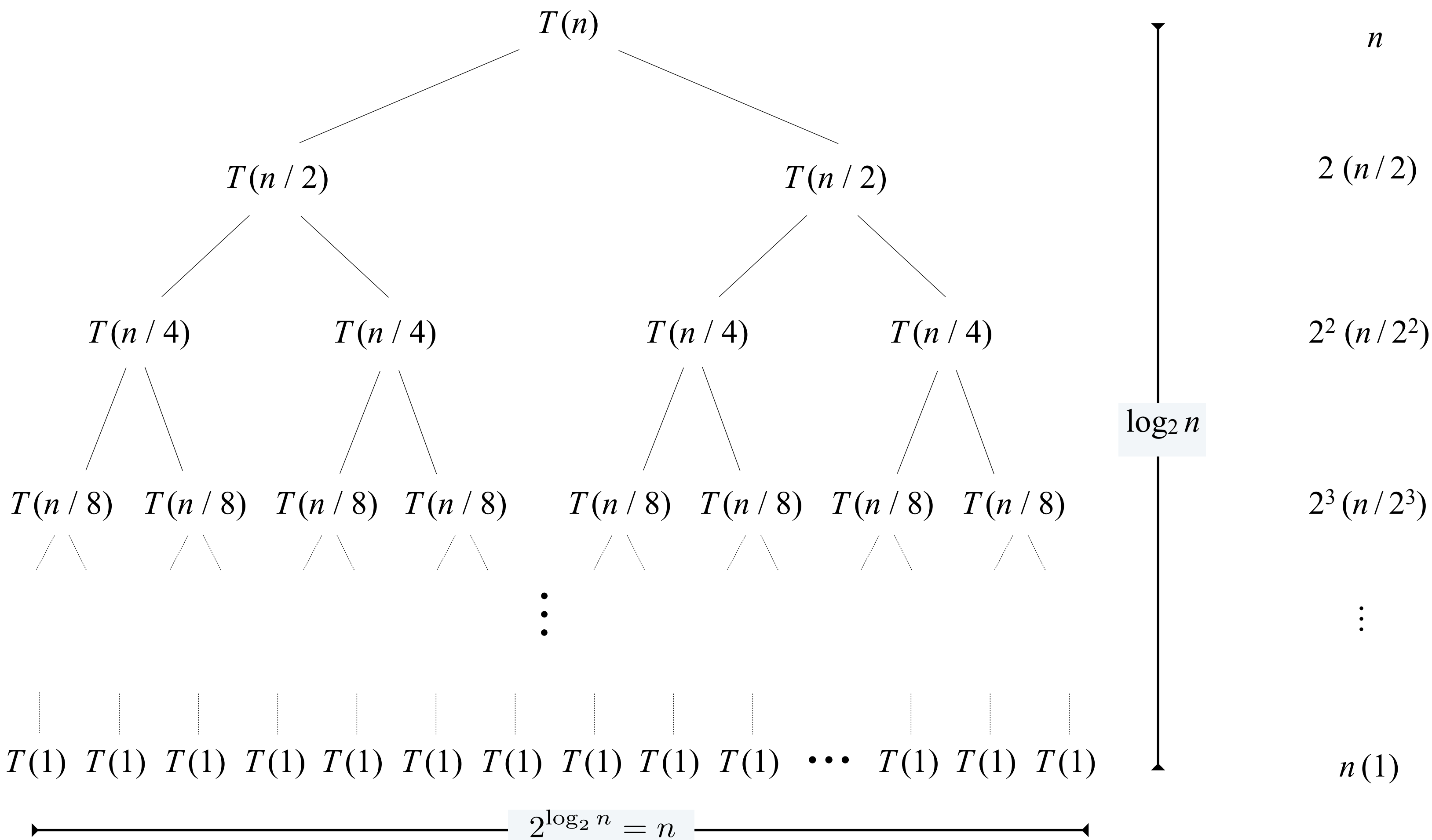
Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $\boldsymbol{T(n) = 2\,T(n/2) + n}$. Then $\boldsymbol{T(n) = \Theta(n \log n)}$.



Case 2: Total cost evenly distributed on all levels

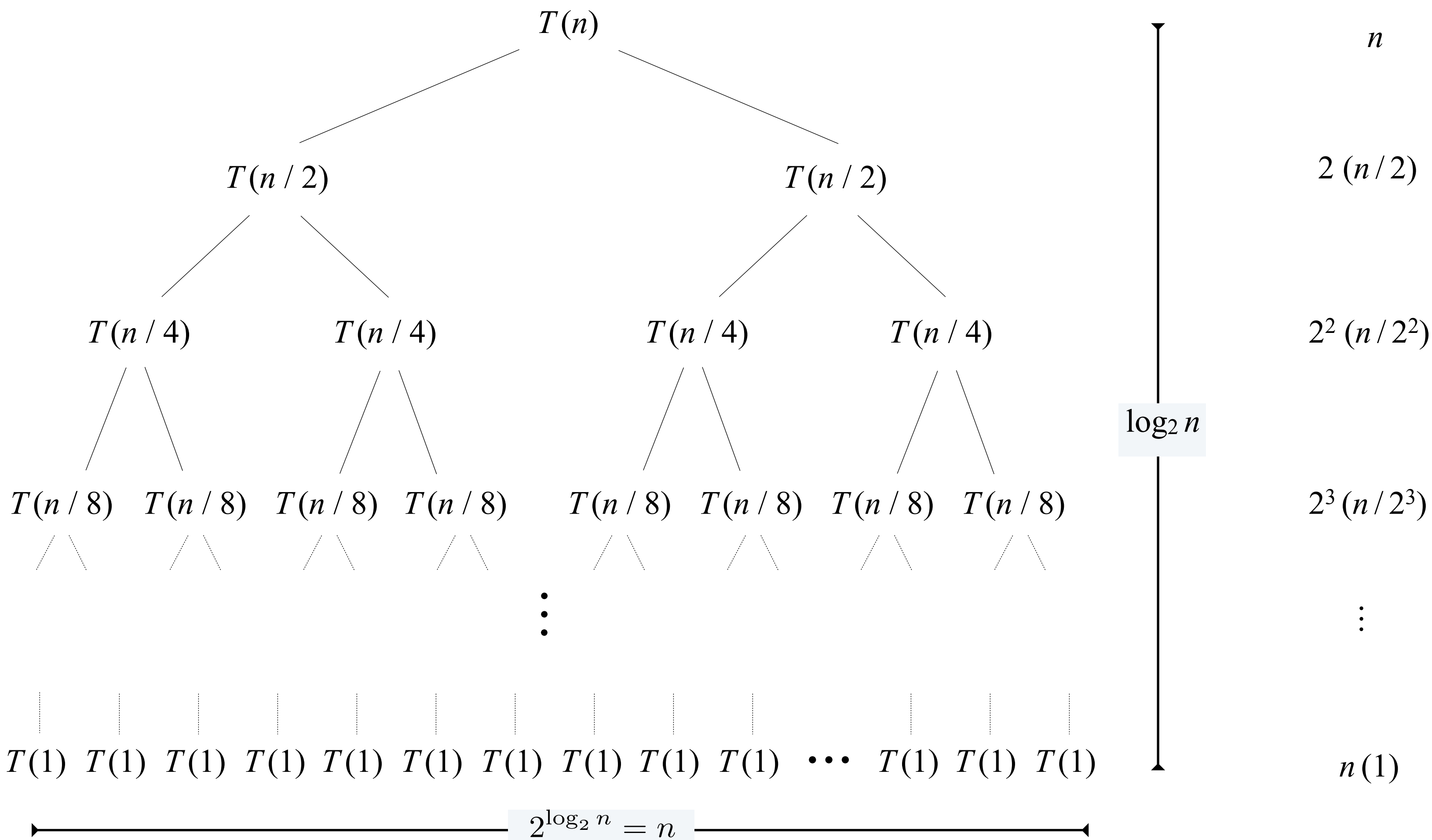
Ex 2. T(1) = 1. $T(n) = 2\,T(n/2) + n$. Then $T(n) = \Theta(n \log n)$.



$r = 1$

Case 2: Total cost evenly distributed on all levels

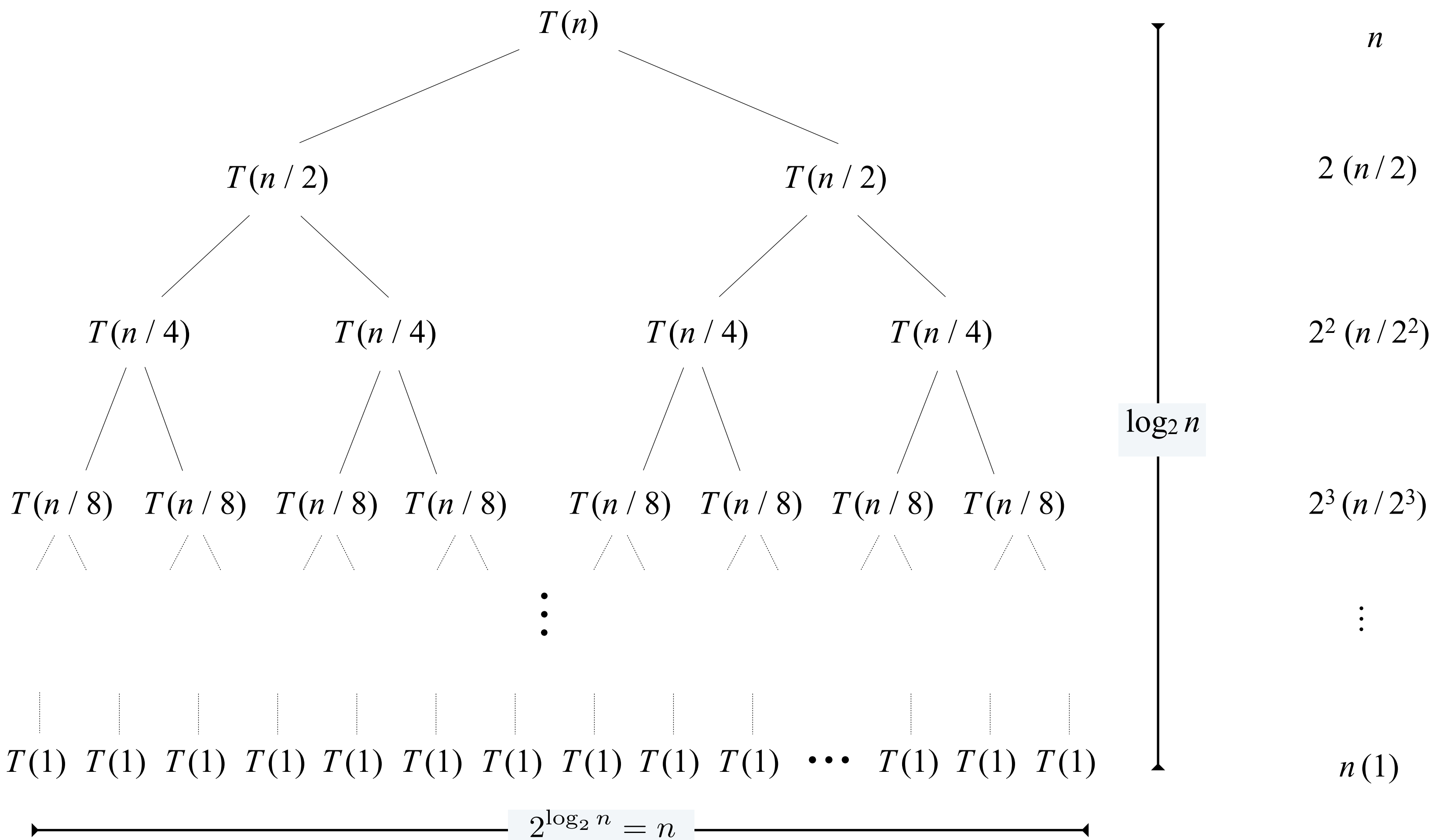
Ex 2. T(1) = 1. $T(n) = 2\,T(n/2) + n$. Then $T(n) = \Theta(n \log n)$.



$r = 1$ $\qquad T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\,n$

Case 2: Total cost evenly distributed on all levels

Ex 2. T(1) = 1. $T(n) = 2\,T(n/2) + n$. Then $T(n) = \Theta(n \log n)$.



$r = 1$ $$T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\, n = n\,(\log_2 n + 1)$$

Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.

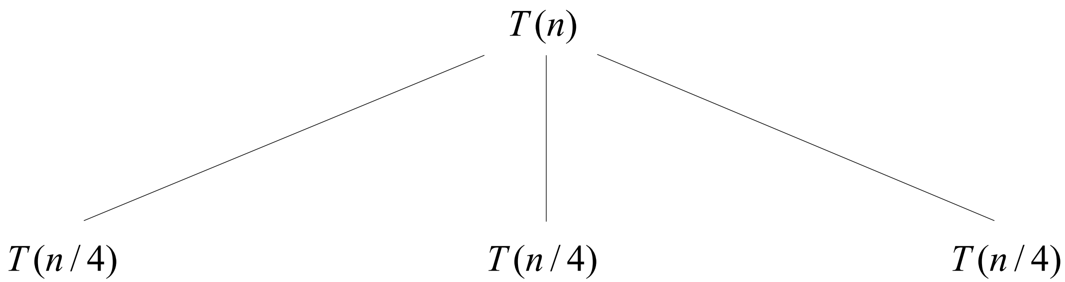
Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\ T(n / 4) + n^5$. Then $T(n) = \Theta(n^5)$.

$T(n)$

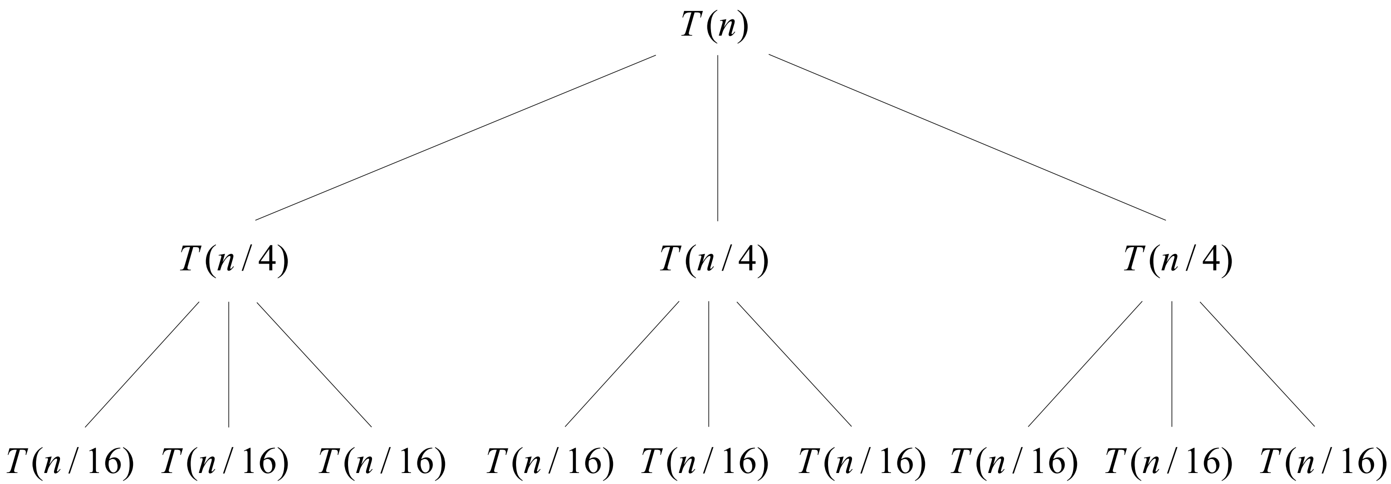
Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



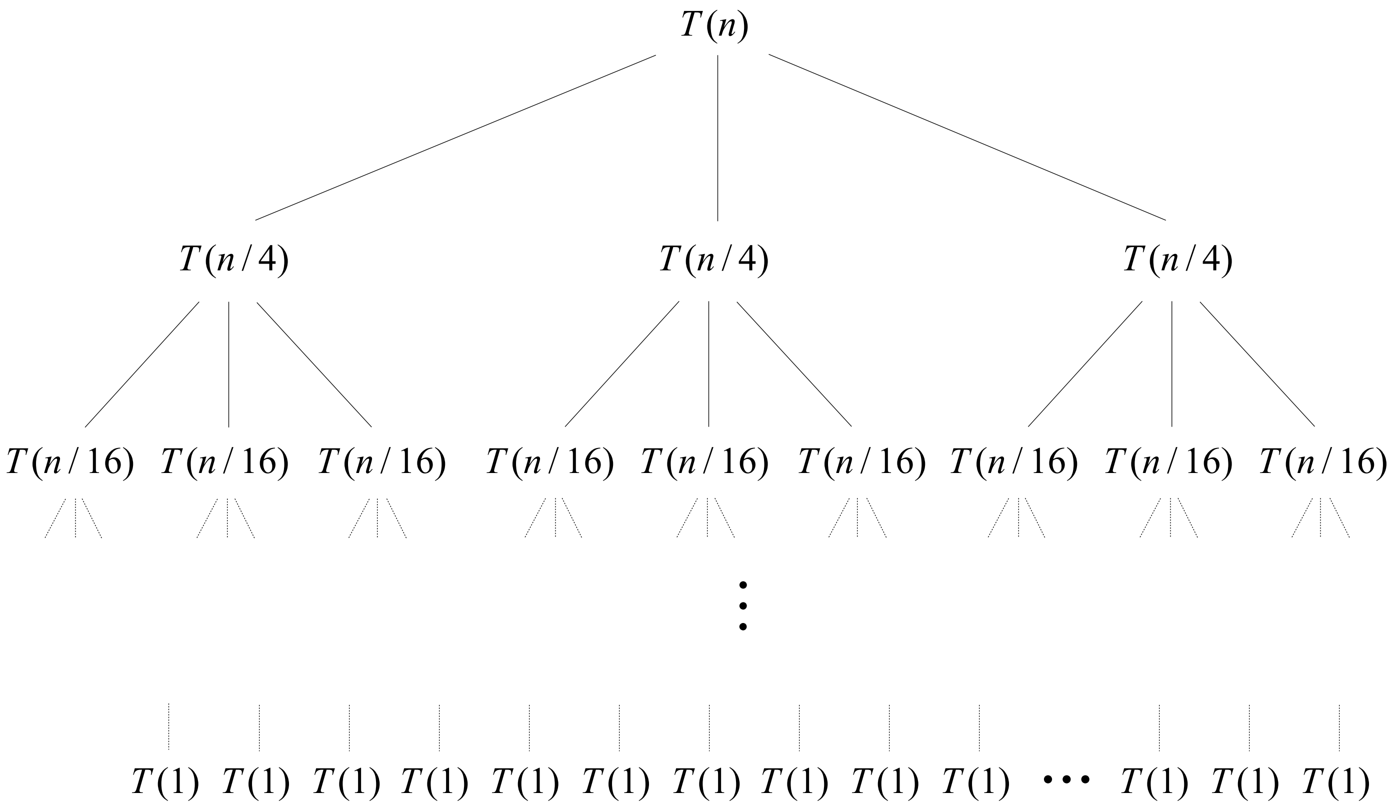
Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



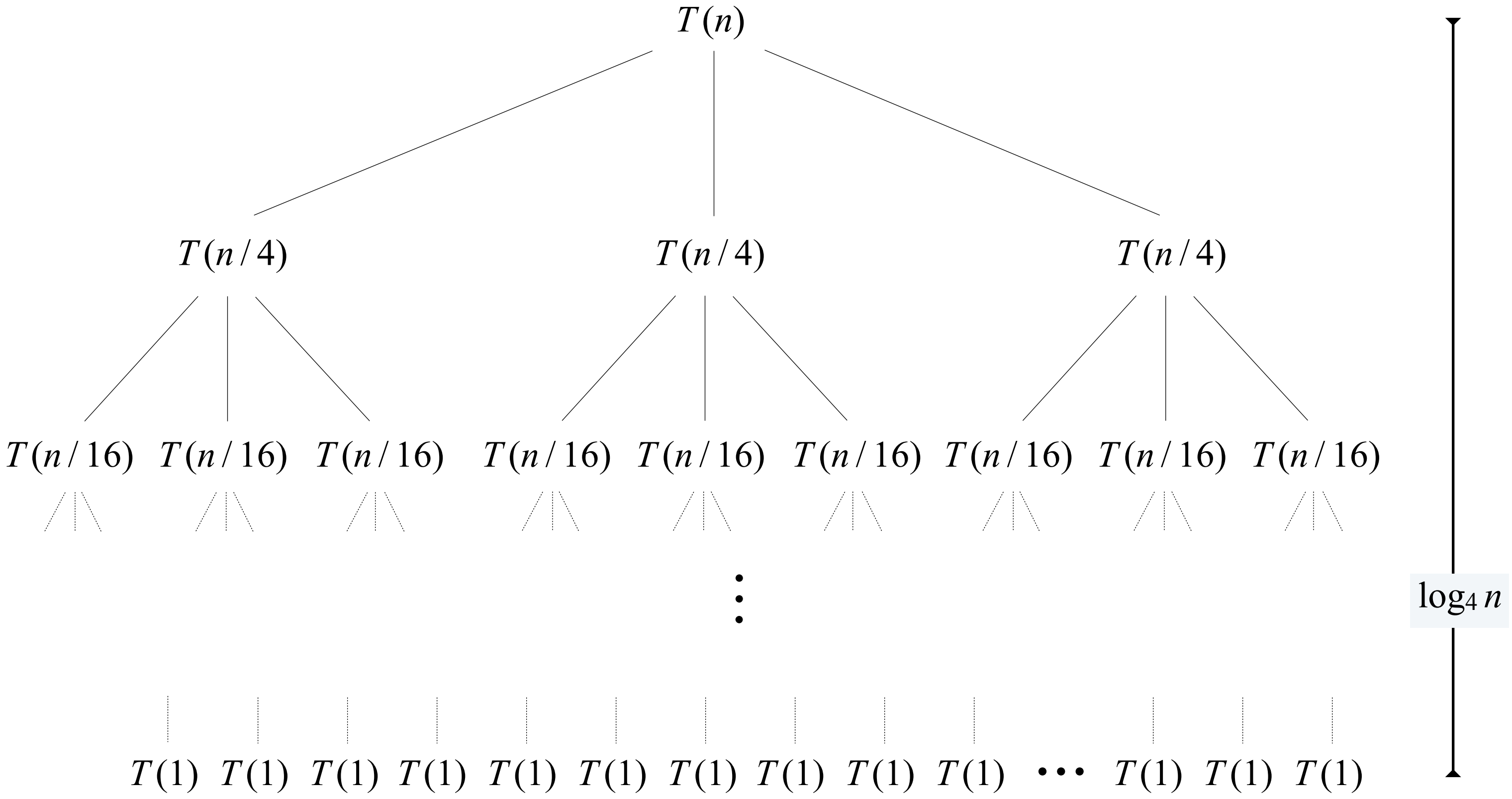
Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



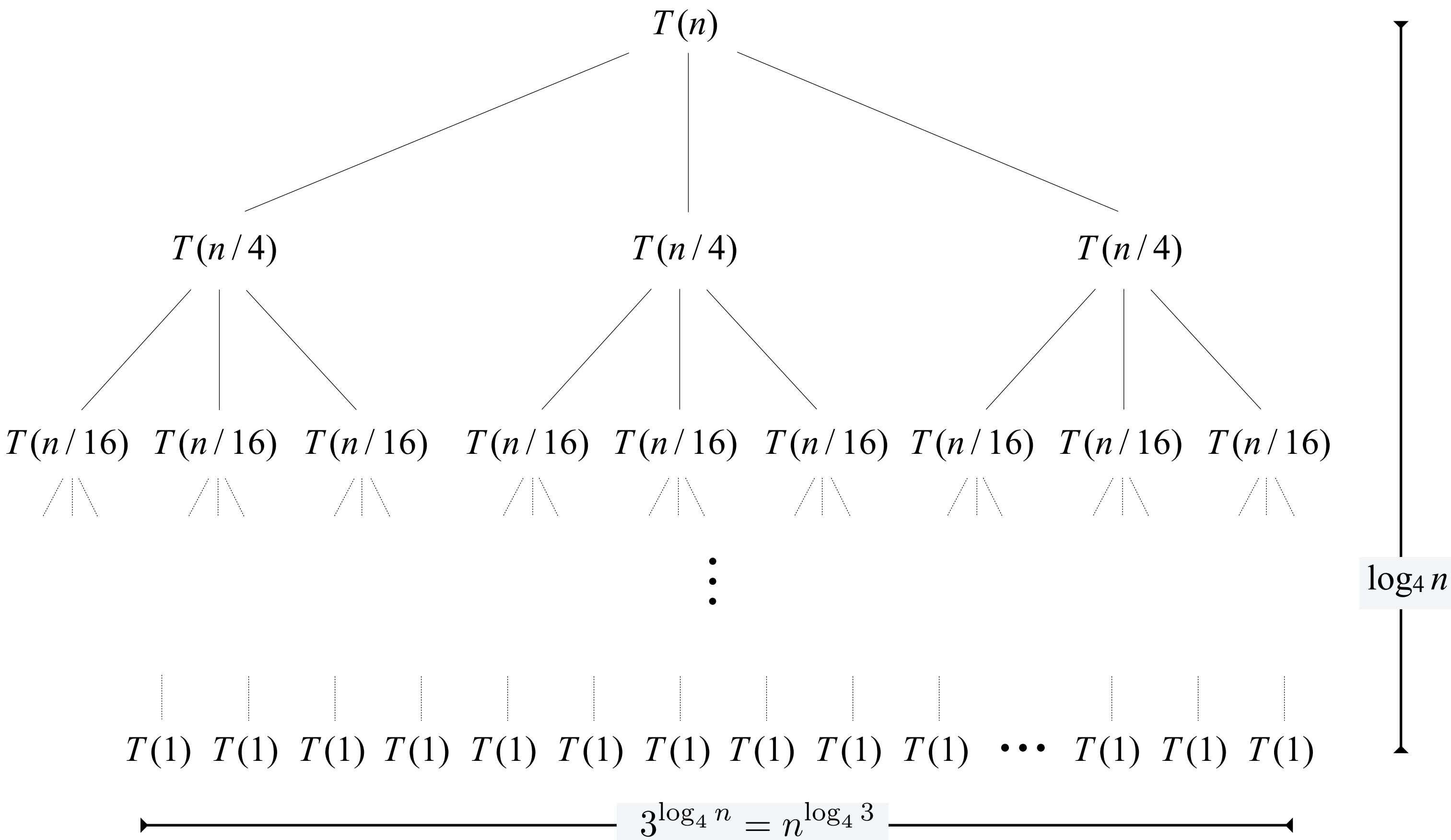
Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



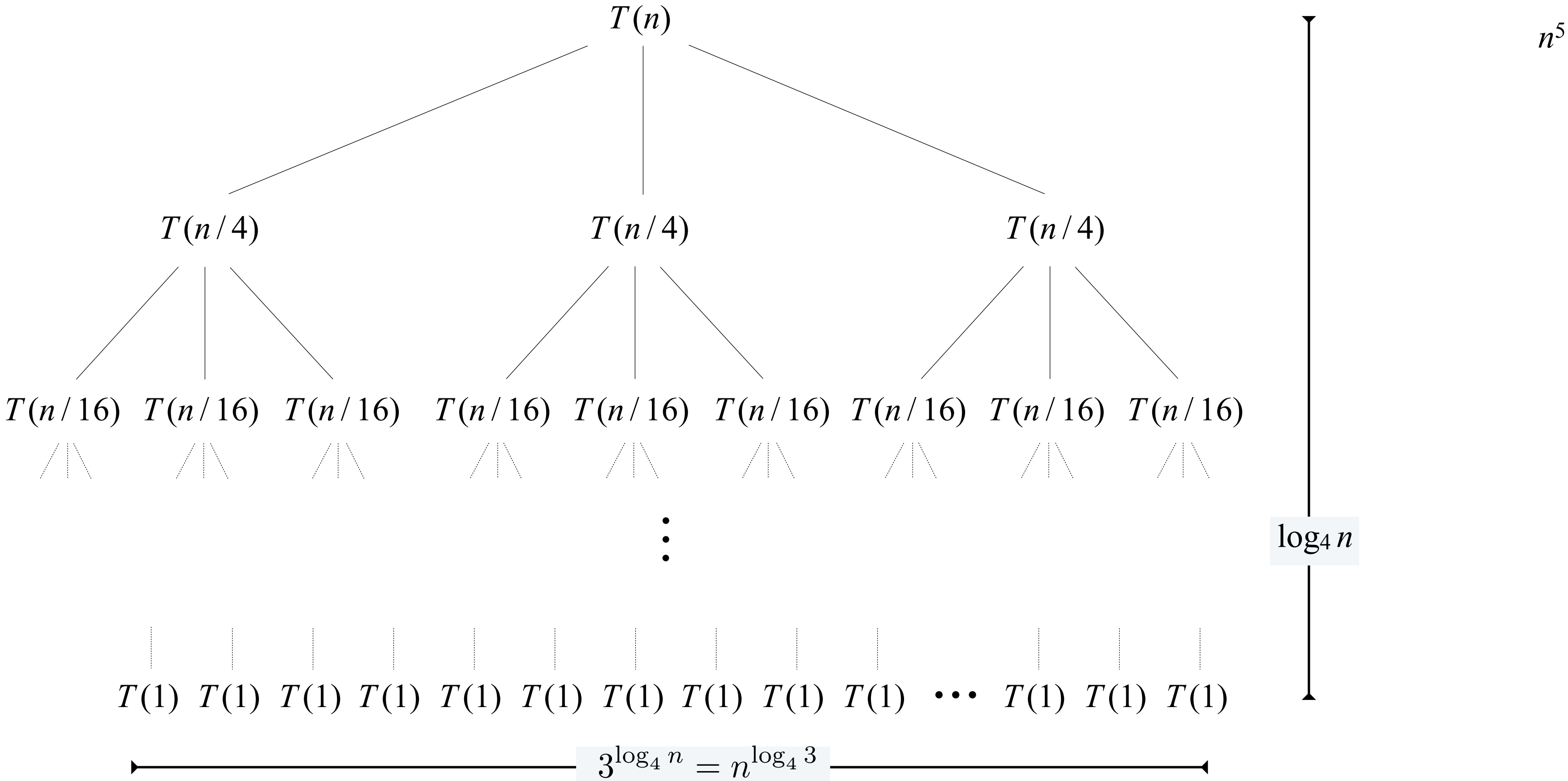
Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



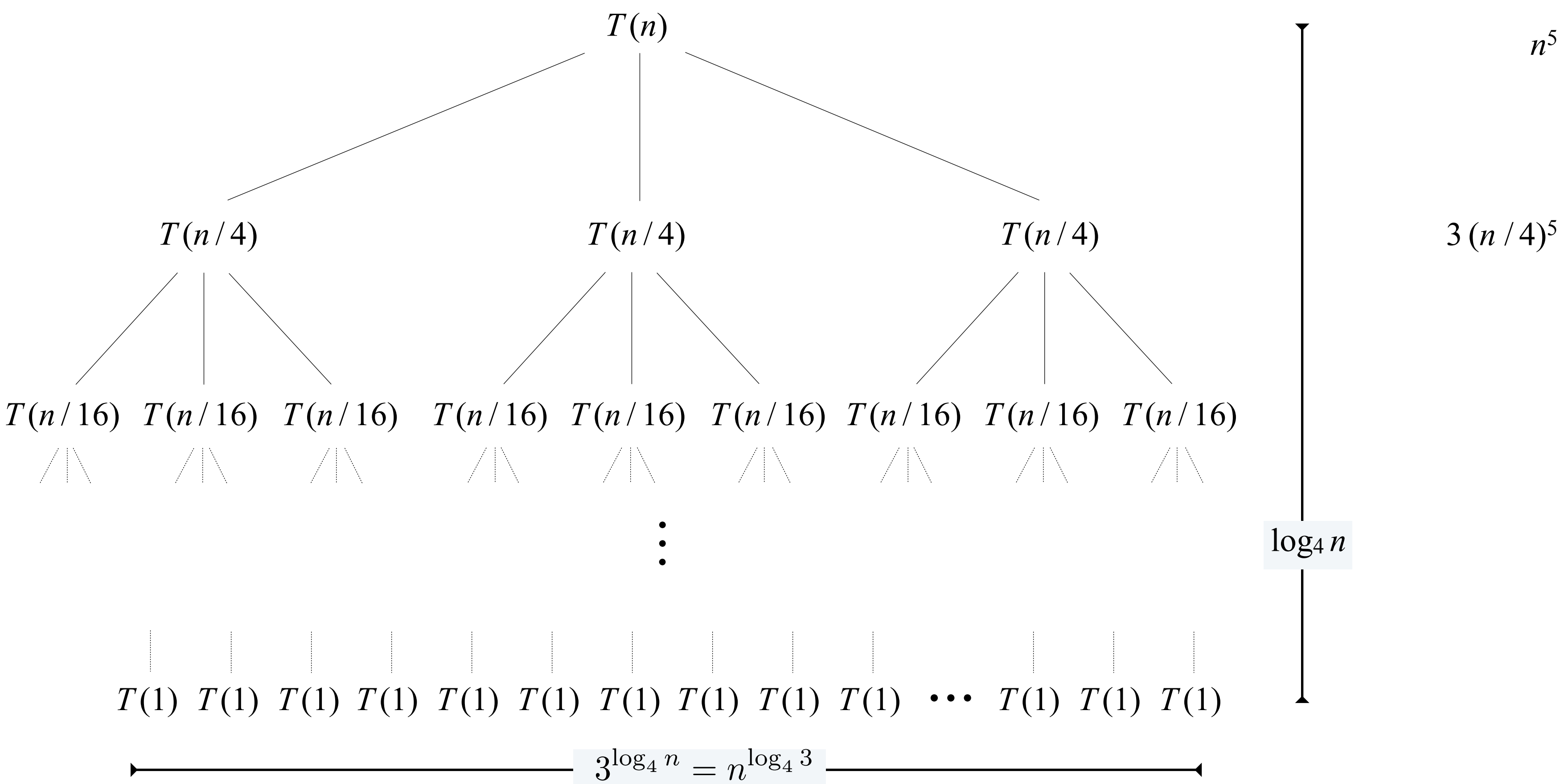
Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



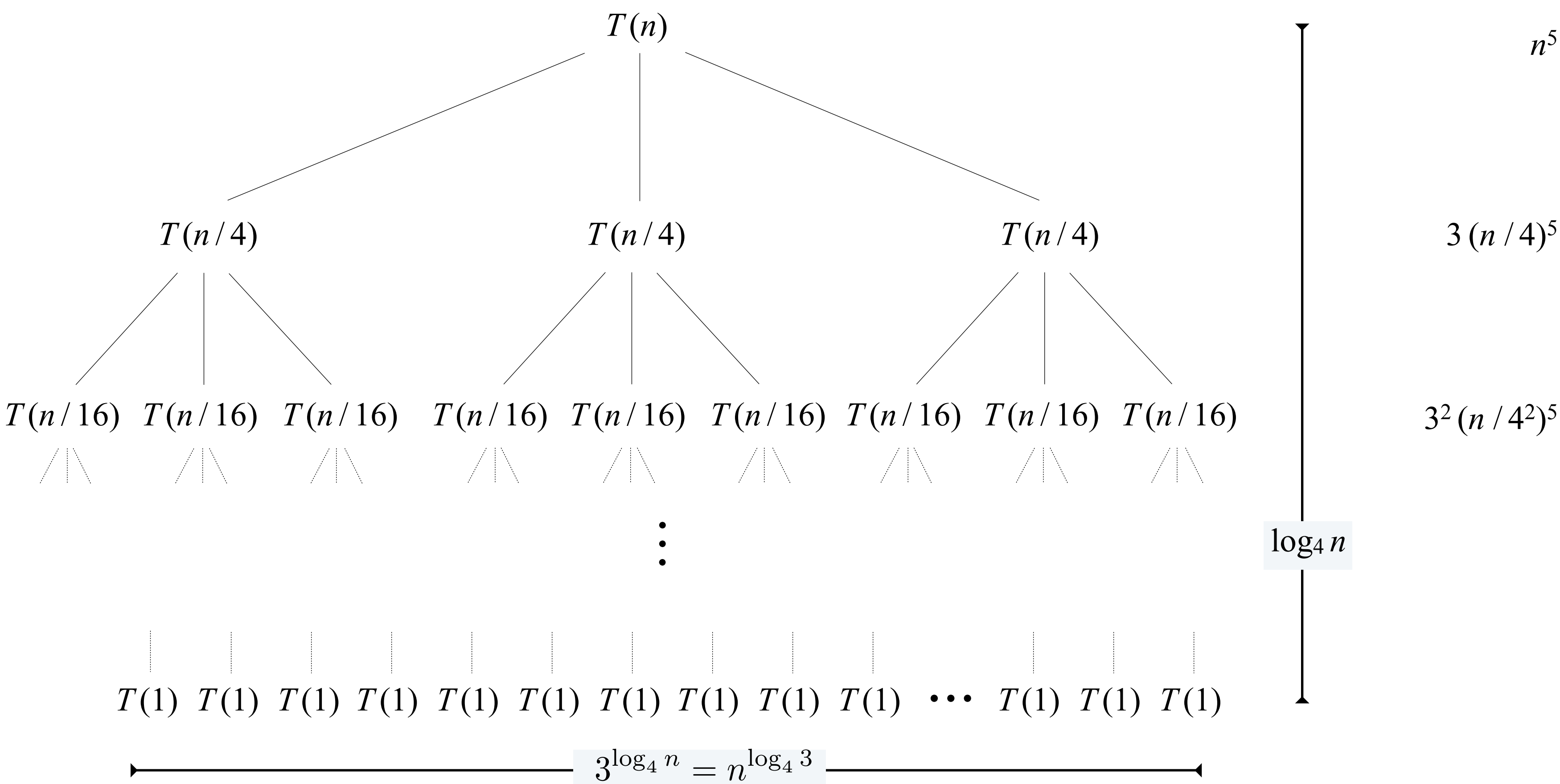
Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\ T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



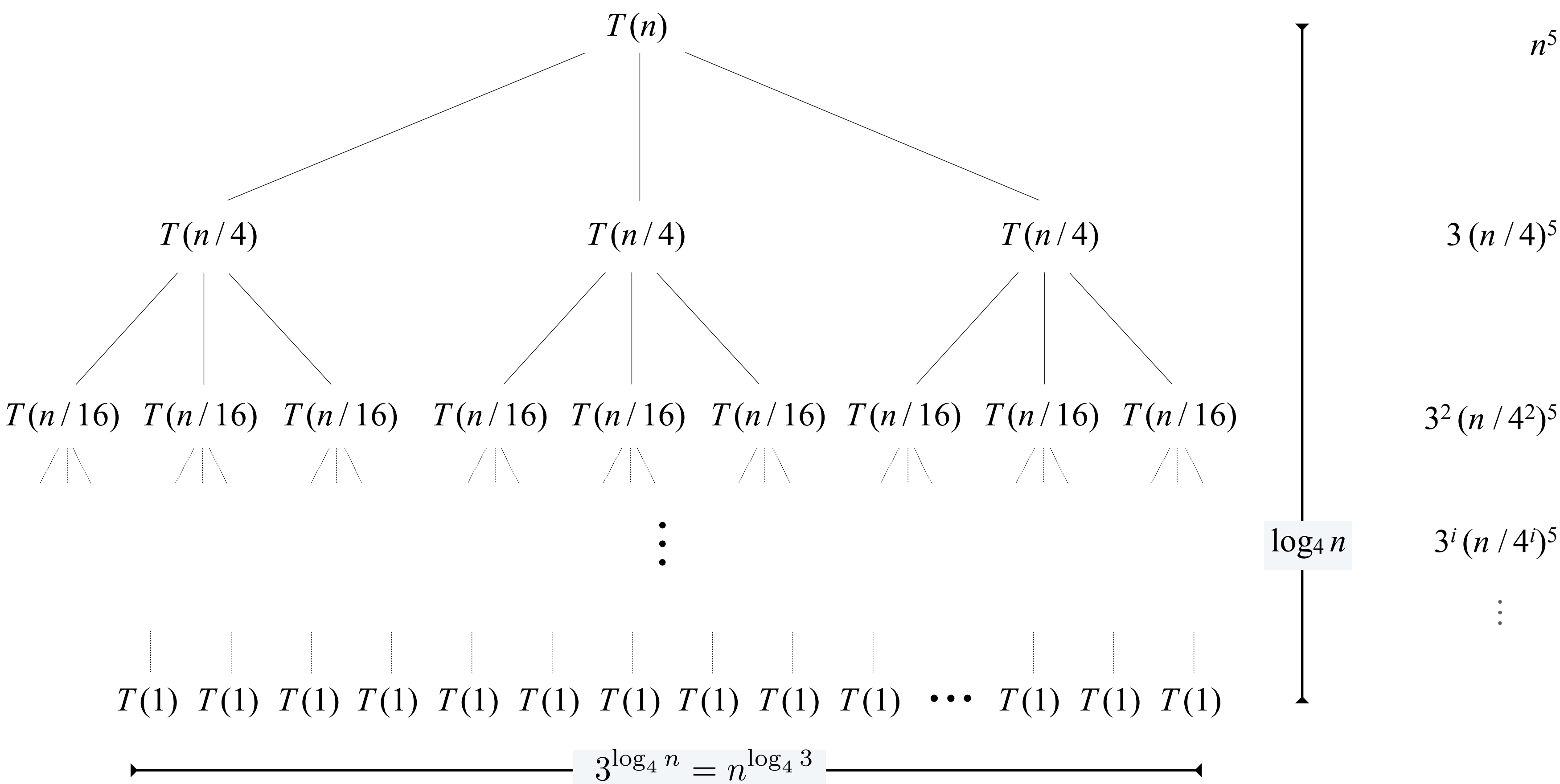
Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



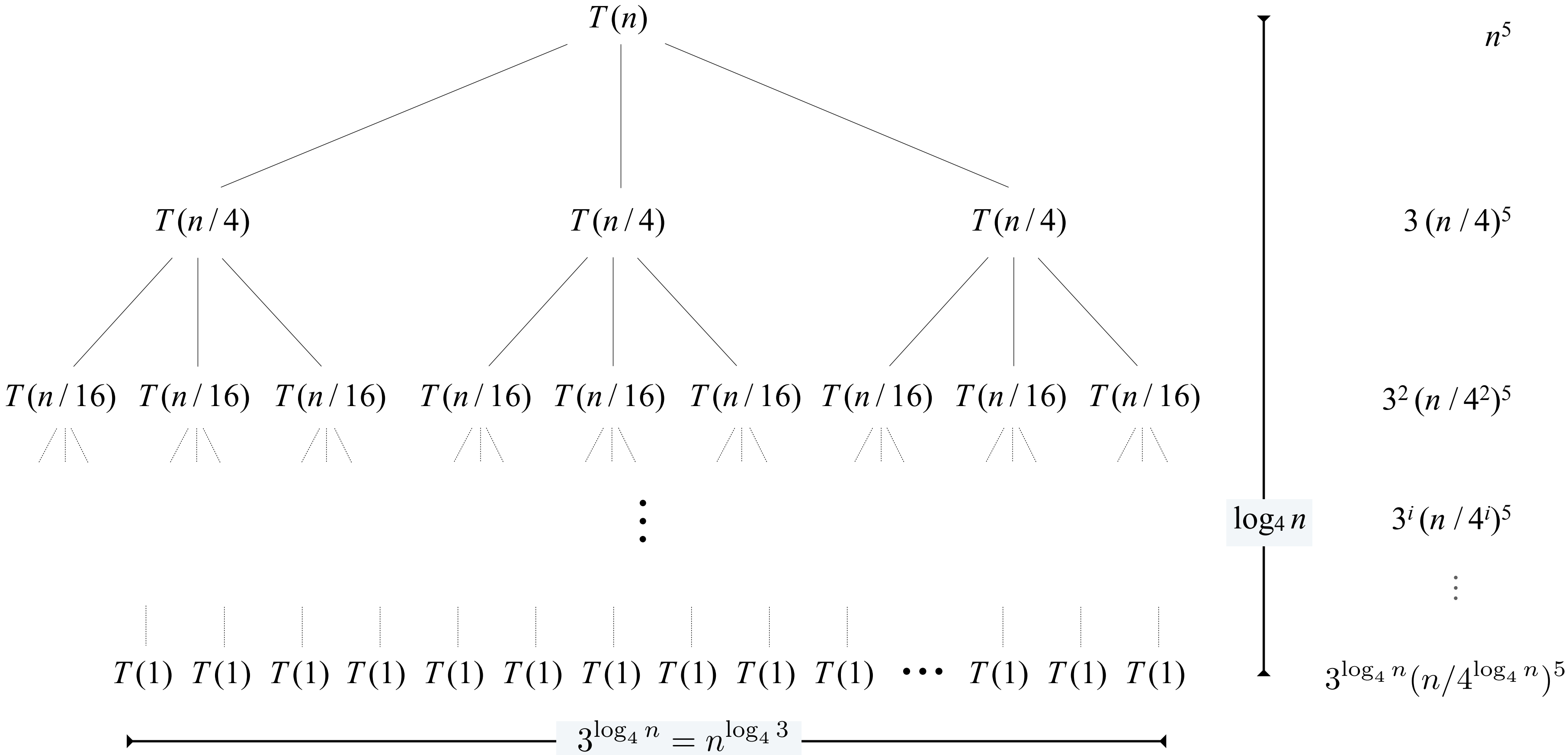
Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



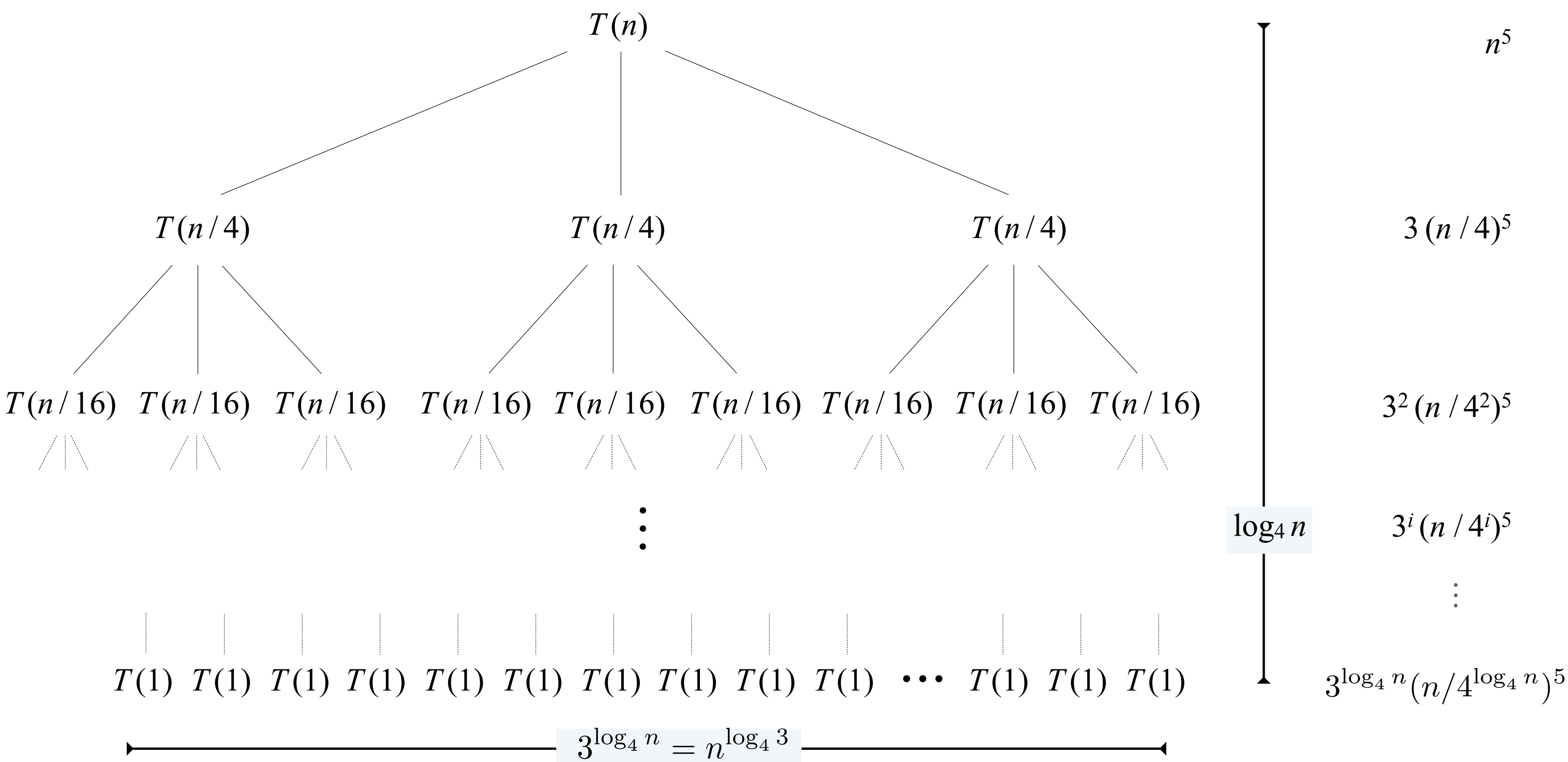
Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



Case 3: Total cost dominated by cost at root

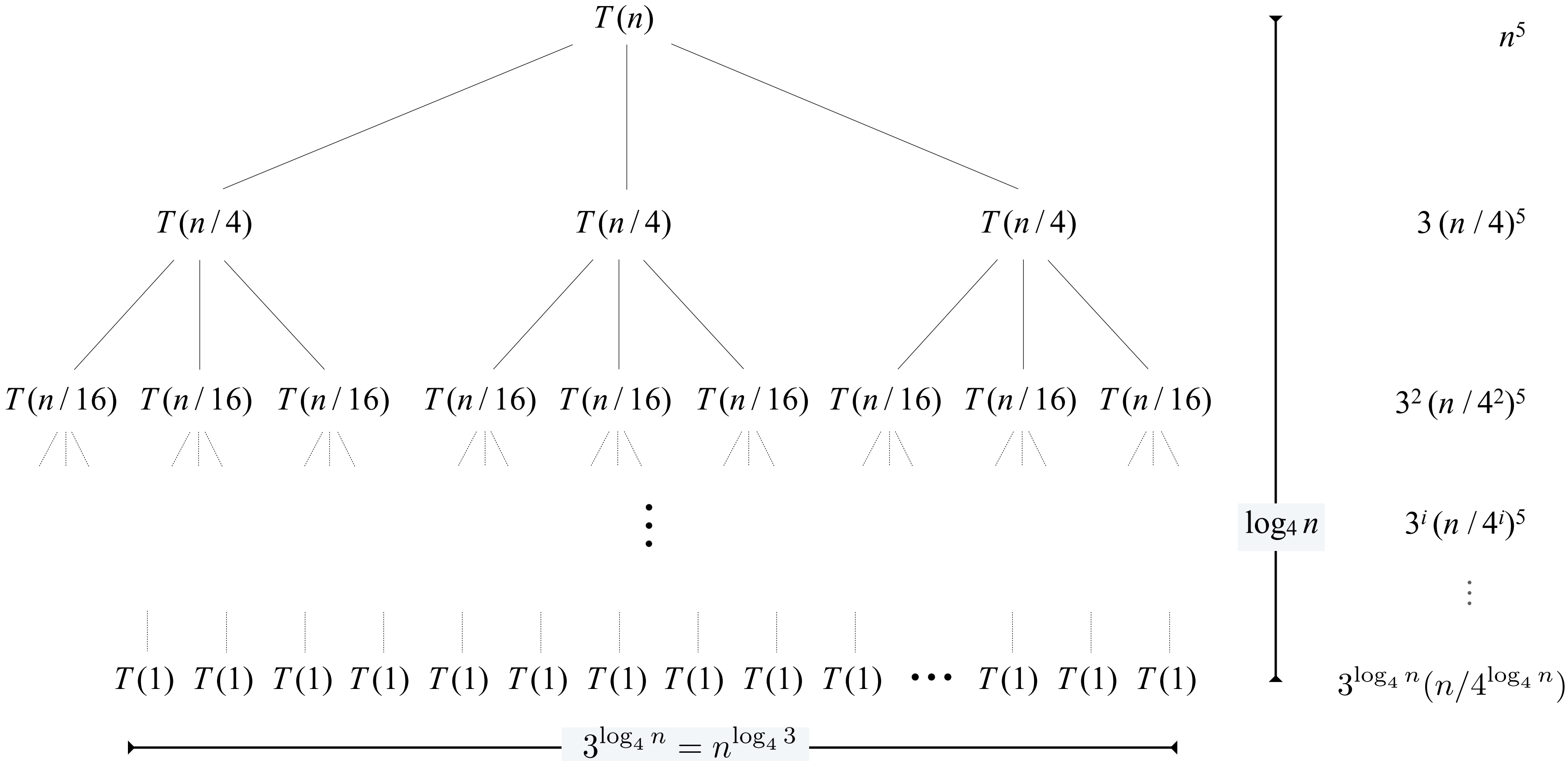
Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



$r = 3 / 4^5 < 1$

Case 3: Total cost dominated by cost at root

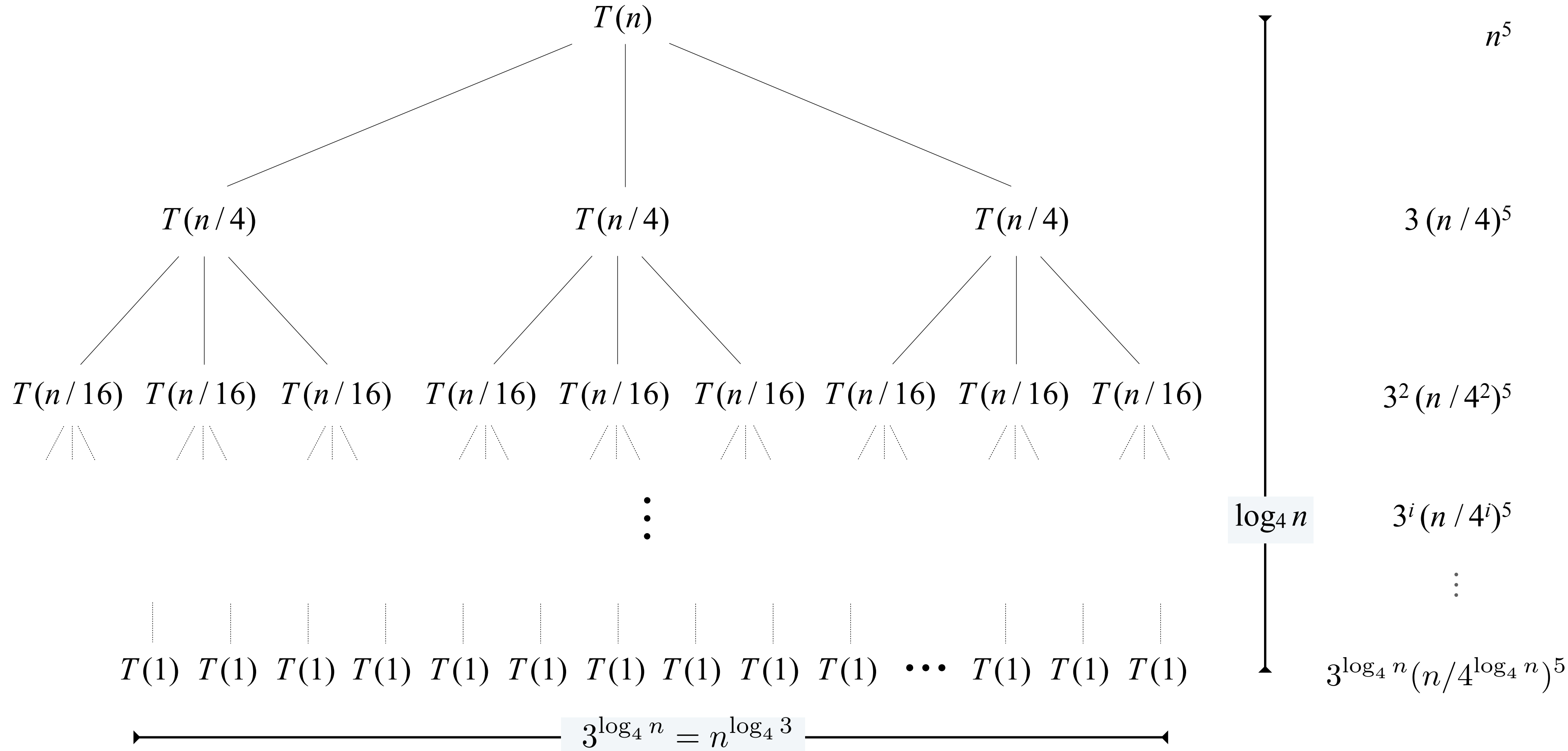
Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



$r = 3/4^5 < 1$ $\quad n^5 \le T(n) = (1 + r + r^2 + r^3 + \ldots)\, n^5 \le$

Case 3: Total cost dominated by cost at root

Ex 2. T(1) = 1. $T(n) = 3\,T(n/4) + n^5$. Then $T(n) = \Theta(n^5)$.



$r = 3 / 4^5 < 1$ $\quad n^5 \le T(n) = (1 + r + r^2 + r^3 + \ldots)\, n^5 \le \dfrac{1}{1-r}\, n^5$

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) \;+\; f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) \;+\; f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

CASE 1: Master theorem
If f(n) = $O(n^{k-\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^k)$

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) \;+\; f(n)}$$

(n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$). Let $k = \log_b a$.

CASE 1: Master theorem
If f(n) = $O(n^{k-\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^k)$

Example: $T(n) = 3\,T(n/2) + n.$

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) \ + \ f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

CASE 1: Master theorem
If f(n) = $O(n^{k-\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^k)$

Example: $T(n) = 3\,T(n / 2) + n$.

- $a = 3,\ b = 2,\ f(n) = n,\ k = \log_2 3.$

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

CASE 1: Master theorem
If f(n) = $O(n^{k-\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^k)$

Example: $T(n) = 3\,T(n / 2) + n$.

- $a = 3,\ b = 2,\ f(n) = n,\ k = \log_2 3$.
- $T(n) = \Theta(n^{\lg 3})$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) \;+\; f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

(n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$). Let $k = \log_b a$.

CASE 2: Master theorem
If f(n) = $O(n^k \log^p n)$, then $T(n) = \Theta(n^k \log^{p+1} n)$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

CASE 2: Master theorem
If f(n) = $O(n^k \log^p n)$, then $T(n) = \Theta(n^k \log^{p+1} n)$.

Example: $T(n) = 2\,T(n / 2) + \Theta(n \log n)$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

CASE 2: Master theorem
If f(n) = $O(n^k \log^p n)$, then $T(n) = \Theta(n^k \log^{p+1} n)$.

Example: $T(n) = 2\,T(n / 2) + \Theta(n \log n)$.

- $a = 2,\ b = 2,\ k = \log_2 2 = 1,\ p = 1$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

CASE 2: Master theorem
If f(n) = $O(n^k \log^p n)$, then $T(n) = \Theta(n^k \log^{p+1} n)$.

Example: $T(n) = 2\,T(n/2) + \Theta(n \log n)$.

- $a = 2,\ b = 2,\ k = \log_2 2 = 1,\ p = 1.$
- $T(n) = \Theta(n \log^2 n)$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) \;+\; f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

(n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$). Let $k = \log_b a$.

CASE 3: Master theorem
If f(n) = $O(n^{k+\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(f(n))$

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

CASE 3: Master theorem
If f(n) = $O(n^{k+\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(f(n))$

Example: $T(n) = 3\,T(n / 4) + n^5$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) \;+\; f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

CASE 3: Master theorem
If f(n) = $O(n^{k+\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(f(n))$

Example: $T(n) = 3\,T(n / 4) + n^5$.

- $a = 3,\; b = 4,\; f(n) = n^5,\; k = \log_4 3$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

CASE 3: Master theorem
If f(n) = $O(n^{k+\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(f(n))$

Example: $T(n) = 3\,T(n / 4) + n^5$.

- $a = 3,\ b = 4,\ f(n) = n^5,\ k = \log_4 3$.
- $T(n) = \Theta(n^5)$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) \;+\; f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

Case 1: If $f(n) = O(n^{k-\varepsilon})$ for some constant $\varepsilon > 0$,
then $T(n) = \Theta(n^k)$.

Case 2: If $f(n) = \Theta(n^k \log^p n)$, then $T(n) = \Theta(n^k \log^{p+1} n)$.

Case 3: If $f(n) = \Omega(n^{k+\varepsilon})$ for some constant $\varepsilon > 0$,
then $T(n) = \Theta(f(n))$.

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers satisfying the recurrence

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

(n / b means either $\lfloor n / b \rfloor$ or $\lceil n / b \rceil$). Let $k = \log_b a$.

Case 1: If $f(n) = O(n^{k-\varepsilon})$ for some constant $\varepsilon > 0$,
then $T(n) = \Theta(n^k)$.

Case 2: If $f(n) = \Theta(n^k \log^p n)$, then $T(n) = \Theta(n^k \log^{p+1} n)$.

Case 3: If $f(n) = \Omega(n^{k+\varepsilon})$ for some constant $\varepsilon > 0$,
then $T(n) = \Theta(f(n))$.

Proof sketch: Recursion tree, case analysis.

Plan for Today

Master theorem

Integer multiplication

Exponentiation

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1

Grade-school algorithm:

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1

Grade-school algorithm:

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
								0

Grade-school algorithm:

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

							1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
							1	0

Grade-school algorithm:

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

						0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
						0	1	0

Grade-school algorithm:

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: $(a + b)$ in binary format.

					1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
					0	0	1	0

Grade-school algorithm:

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

				1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
				1	0	0	1	0

Grade-school algorithm:

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

			1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
			0	1	0	0	1	0

Grade-school algorithm:

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

		1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
		1	0	1	0	0	1	0

Grade-school algorithm:

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

	1	1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
	0	1	0	1	0	0	1	0

Grade-school algorithm:

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

1	1	1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

Grade-school algorithm:

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

1	1	1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

Grade-school algorithm:

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

1	1	1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

Grade-school algorithm: O(n) operations.

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a - b) in binary format.

	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a - b) in binary format.

	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1

Grade-school algorithm:

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a - b) in binary format.

	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1

Grade-school algorithm:

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a - b) in binary format.

	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
								0

Grade-school algorithm:

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: $(a - b)$ in binary format.

	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
							0	0

Grade-school algorithm:

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a - b) in binary format.

						0		
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
						0	0	0

Grade-school algorithm:

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a - b) in binary format.

					+	0		
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
					1	0	0	0

Grade-school algorithm:

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: $(a - b)$ in binary format.

				+	+	0		
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
				1	1	0	0	0

Grade-school algorithm:

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: $(a - b)$ in binary format.

			1	+	+	0		
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
			0	1	1	0	0	0

Grade-school algorithm:

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a - b) in binary format.

		+	1	+	+	0		
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
		1	0	1	1	0	0	0

Grade-school algorithm:

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a - b) in binary format.

	1	+	1	+	+	0		
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
	0	1	0	1	1	0	0	0

Grade-school algorithm:

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a - b) in binary format.

	1	+	1	+	+	0		
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	1	0	0	0

Grade-school algorithm:

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a - b) in binary format.

	1	+	1	+	+	0		
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	1	0	0	0

Grade-school algorithm:

Basic arithmetic ops

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a - b) in binary format.

	1	+	1	+	+	0		
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	1	0	0	0

Grade-school algorithm: O(n) operations.

Basic arithmetic ops

Addition
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a + b) in binary format.

Subtraction
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a - b) in binary format.

Grade-school algorithms: $\Theta(n)$ operations.

Asymptotically optimal.

Basic arithmetic ops

Multiplication
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a x b) in binary format.

	1	1	0	1	0	1	0	1
x	0	1	1	1	1	1	0	1
?	?	?	?	?	?	?	?	?

Grade-school algorithm:

Basic arithmetic ops

Multiplication
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a x b) in binary format.

	1	1	0	1	0	1	0	1
x	0	1	1	1	1	1	0	1
?	?	?	?	?	?	?	?	?

Grade-school algorithm: $O(n^2)$ operations.

Basic arithmetic ops

Multiplication
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a x b) in binary format.

	1	1	0	1	0	1	0	1
x	0	1	1	1	1	1	0	1
?	?	?	?	?	?	?	?	?

Grade-school algorithm: $O(n^2)$ operations.

Conjecture [Kolmogorov 1952]: This is optimal!

Basic arithmetic ops

Multiplication
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a x b) in binary format.

	1	1	0	1	0	1	0	1
x	0	1	1	1	1	1	0	1
?	?	?	?	?	?	?	?	?

Grade-school algorithm: $O(n^2)$ operations.

Basic arithmetic ops

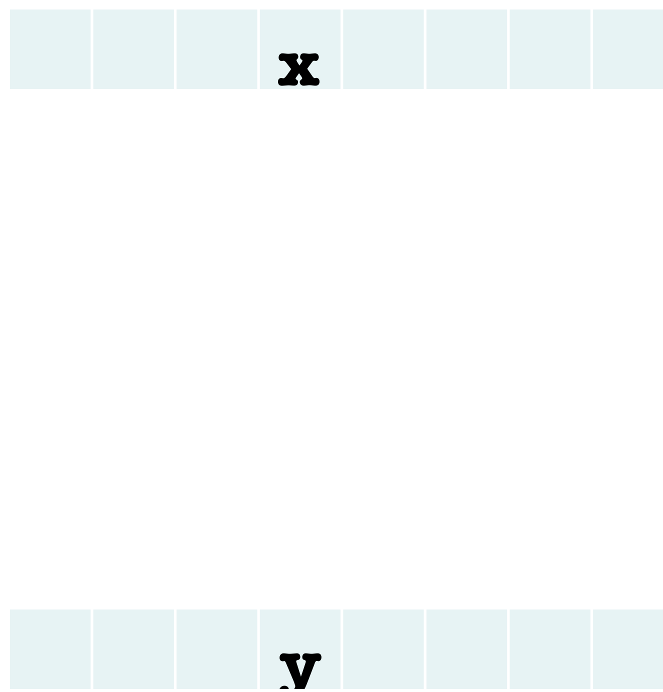
Multiplication
INPUT: Two n-bit numbers a, b in binary
OUTPUT: (a x b) in binary format.

	1	1	0	1	0	1	0	1
x	0	1	1	1	1	1	0	1
?	?	?	?	?	?	?	?	?

Grade-school algorithm: $O(n^2)$ operations.

Theorem [Karatsuba 1960]: Conjecture false!

Divide-and-conquer multiplication

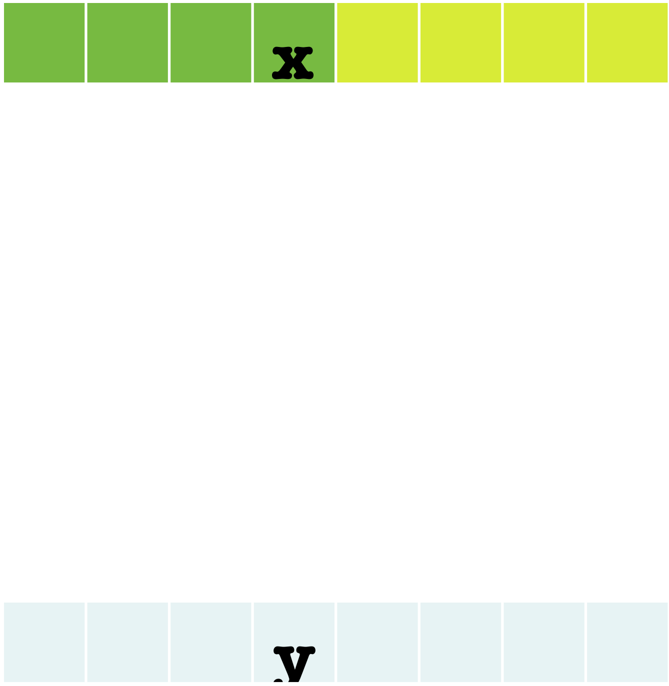


Divide-and-conquer multiplication

1. Divide **x** and **y** into high and low-order bits

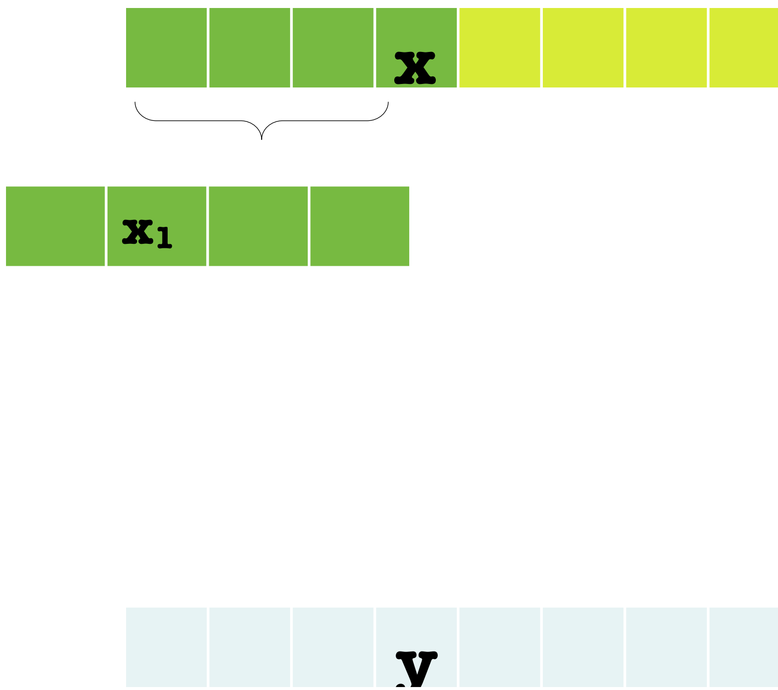
Divide-and-conquer multiplication

1. Divide **x** and **y** into high and low-order bits



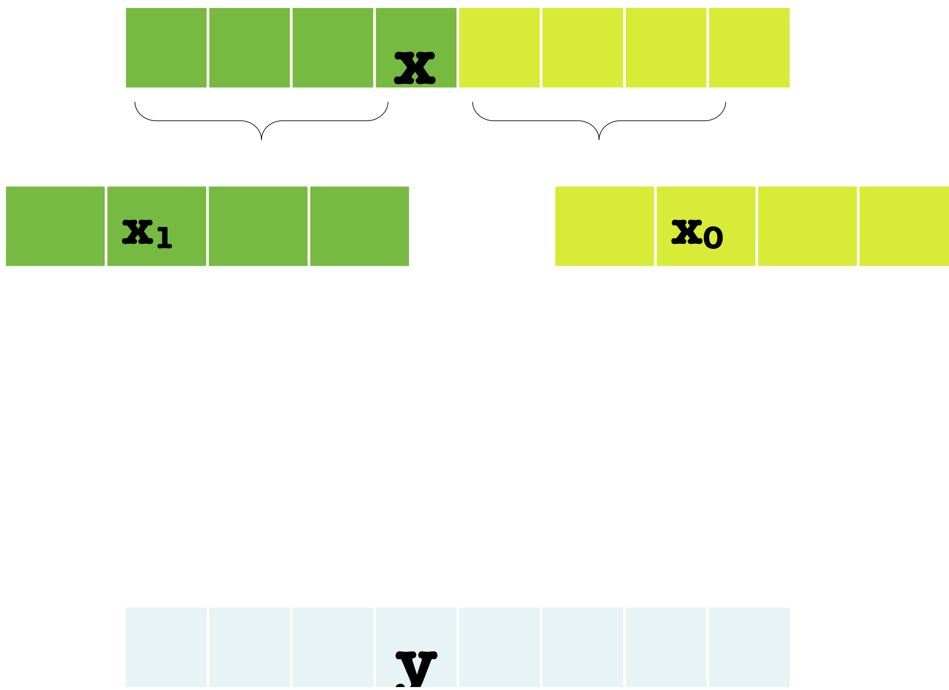
Divide-and-conquer multiplication

1. **Divide $\mathbf{x}$ and $\mathbf{y}$ into high and low-order bits**



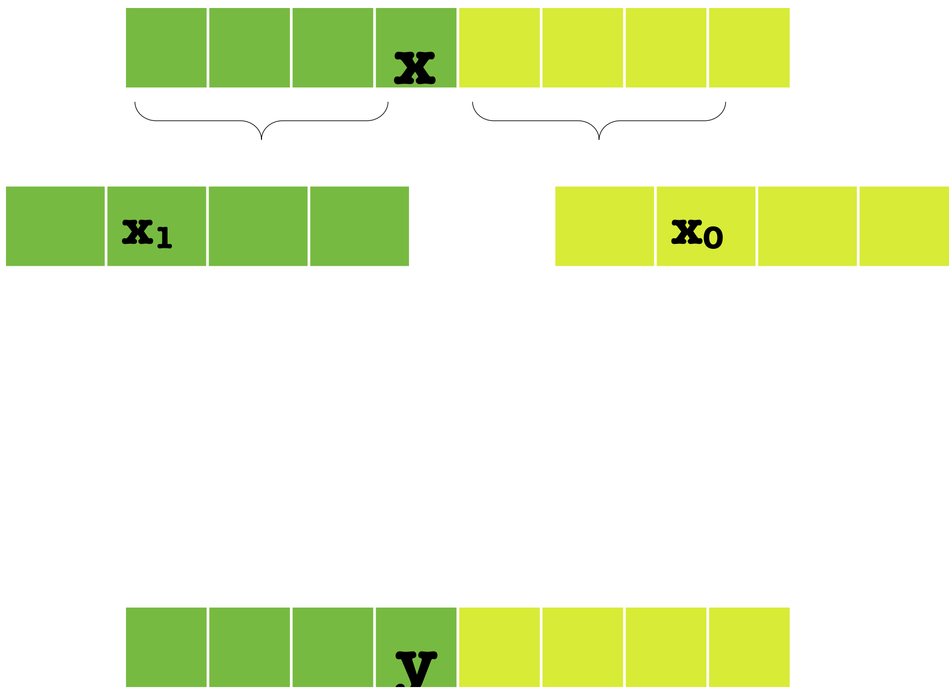
Divide-and-conquer multiplication

1. Divide $\mathbf{x}$ and $\mathbf{y}$ into high and low-order bits



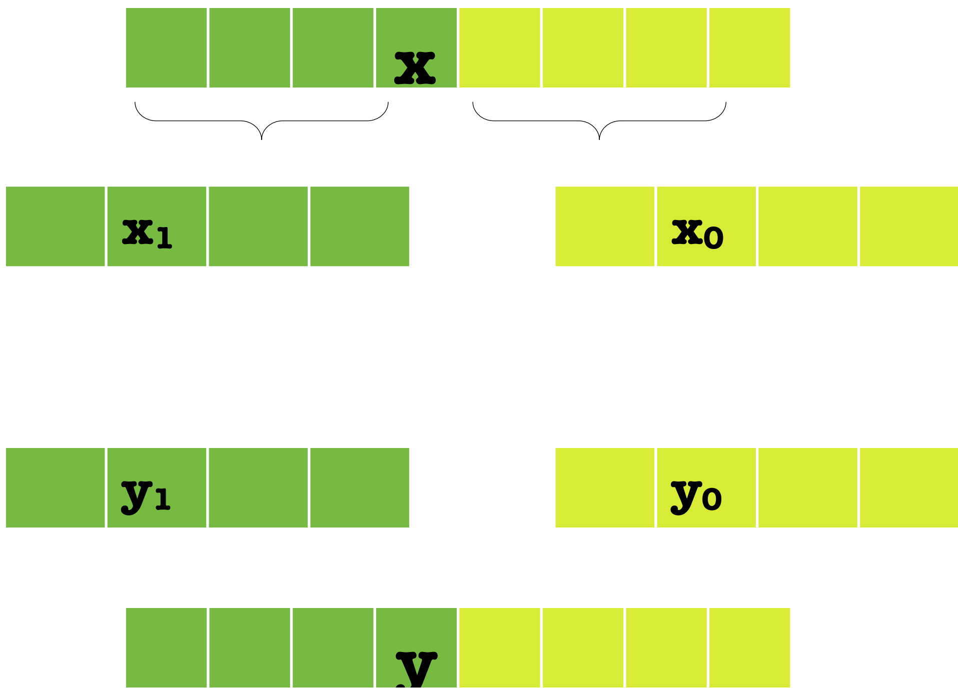
Divide-and-conquer multiplication

1. Divide **x** and **y** into high and low-order bits



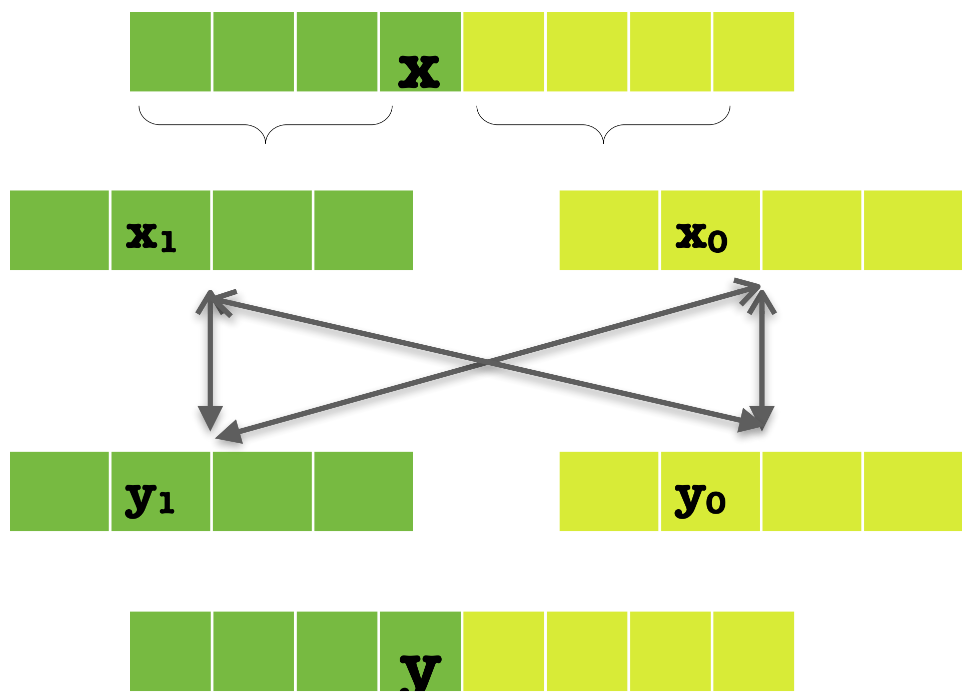
Divide-and-conquer multiplication

1. Divide **x** and **y** into high and low-order bits



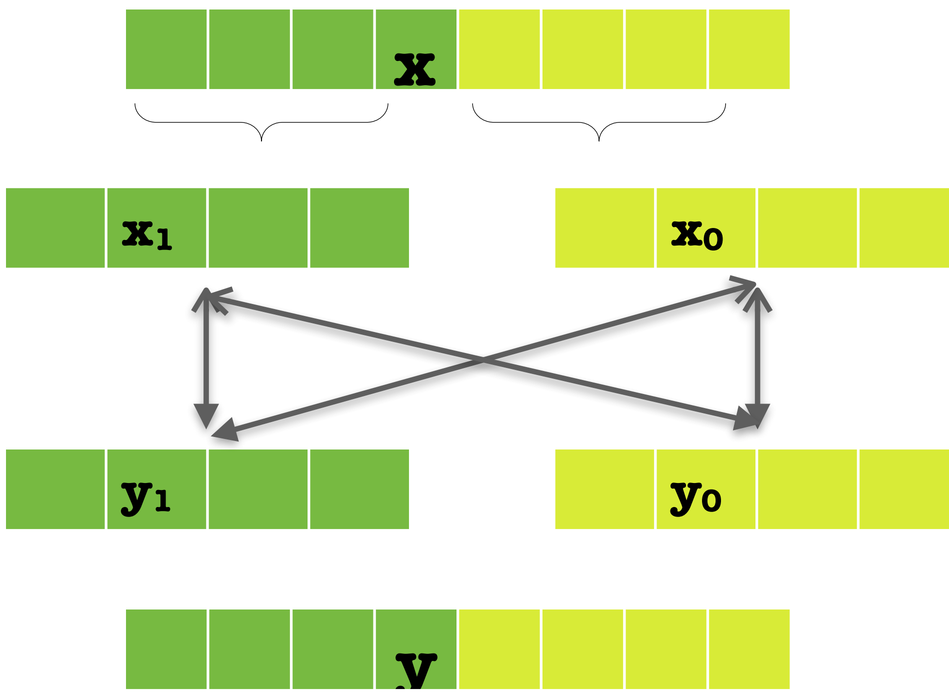
Divide-and-conquer multiplication

1. Divide **x** and **y** into high and low-order bits
2. Multiply four (n/2)-bit integers recursively.



Divide-and-conquer multiplication

1. Divide **x** and **y** into high and low-order bits
2. Multiply four (n/2)-bit integers recursively.
3. Shift and add to obtain **x*y**.



Divide-and-conquer multiplication

1. **Divide `x` and `y` into high and low-order bits**
2. **Multiply four (n/2)-bit integers recursively.**
3. **Shift and add to obtain `x*y`.**

$$x = 2^{n/2}x_1 + x_0$$

Divide-and-conquer multiplication

1. **Divide `x` and `y` into high and low-order bits**
2. **Multiply four (n/2)-bit integers recursively.**
3. **Shift and add to obtain `x*y`.**

$$x = 2^{n/2} x_1 + x_0$$

$$y = 2^{n/2} y_1 + y_0$$

Divide-and-conquer multiplication

1. **Divide `x` and `y` into high and low-order bits**
2. **Multiply four (n/2)-bit integers recursively.**
3. **Shift and add to obtain `x*y`.**

$$x = 2^{n/2}x_1 + x_0$$

$$y = 2^{n/2}y_1 + y_0$$

$$x \cdot y = (2^{n/2}x_1 + x_0) \cdot (2^{n/2}y_1 + y_0)$$

Divide-and-conquer multiplication

1. **Divide `x` and `y` into high and low-order bits**
2. **Multiply four (n/2)-bit integers recursively.**
3. **Shift and add to obtain `x*y`.**

$$
\boxed{
\begin{aligned}
x &= 2^{n/2} x_1 + x_0 \\
y &= 2^{n/2} y_1 + y_0 \\
x \cdot y &= (2^{n/2} x_1 + x_0) \cdot (2^{n/2} y_1 + y_0) \\
&= 2^n (x_1 \cdot y_1) + 2^{n/2}((x_0 \cdot y_1) + (x_1 \cdot y_0)) + (x_0 \cdot y_0)
\end{aligned}
}
$$

Divide-and-conquer multiplication

1. **Divide $\mathbf{x}$ and $\mathbf{y}$ into high and low-order bits**
2. **Multiply four (n/2)-bit integers recursively.**
3. **Shift and add to obtain $\mathbf{x*y}$.**

$$
\begin{aligned}
x &= 2^{n/2}x_1 + x_0 \\
y &= 2^{n/2}y_1 + y_0 \\
x \cdot y &= (2^{n/2}x_1 + x_0) \cdot (2^{n/2}y_1 + y_0) \\
&= 2^n(x_1 \cdot y_1) + 2^{n/2}((x_0 \cdot y_1) + (x_1 \cdot y_0)) + (x_0 \cdot y_0)
\end{aligned}
$$

Ex. $x = \underbrace{1\ 0\ 0\ 0}_{x1}\ \underbrace{1\ 1\ 0\ 1}_{x0} \qquad y = \underbrace{1\ 1\ 1\ 0}_{y1}\ \underbrace{0\ 0\ 0\ 1}_{y0}$

Divide-and-conquer multiplication

Proposition. The divide-and-conquer multiplication algorithm requires $\Theta(n^2)$ bit operations to multiply two n-bit integers.

Divide-and-conquer multiplication

Proposition. The divide-and-conquer multiplication algorithm requires $\Theta(n^2)$ bit operations to multiply two n-bit integers.

$$\boxed{T(n) = \underbrace{4T(\lceil n/2 \rceil)}_{\text{four recursive calls}} + \underbrace{O(n)}_{\text{three additions}}}$$

Divide-and-conquer multiplication

Proposition. The divide-and-conquer multiplication algorithm requires $\Theta(n^2)$ bit operations to multiply two n-bit integers.

$$\boxed{T(n) = \underbrace{4T(\lceil n/2 \rceil)}_{\text{four recursive calls}} + \underbrace{O(n)}_{\text{three additions}}}$$

Pf. Apply first-case of the master theorem to the recurrence: a = 4, b = 2, f(n) = O(n).

Divide-and-conquer multiplication

Proposition. The divide-and-conquer multiplication algorithm requires $\Theta(n^2)$ bit operations to multiply two n-bit integers.

$$T(n) = \underbrace{4T(\lceil n/2 \rceil)}_{\text{four recursive calls}} + \underbrace{O(n)}_{\text{three additions}}$$

Pf. Apply first-case of the master theorem to the recurrence: a = 4, b = 2, f(n) = O(n).

Much ado about nothing??

Divide-and-conquer multiplication

1. **Divide `x` and `y` into high and low-order bits**
2. **Multiply four (n/2)-bit integers recursively.**
3. **Shift and add to obtain `x*y`.**

$$
\begin{aligned}
x &= 2^{n/2}x_1 + x_0 \\
y &= 2^{n/2}y_1 + y_0 \\
x \cdot y &= (2^{n/2}x_1 + x_0) \cdot (2^{n/2}y_1 + y_0) \\
&= 2^n(x_1 \cdot y_1) + 2^{n/2}((x_0 \cdot y_1) + (x_1 \cdot y_0)) + (x_0 \cdot y_0)
\end{aligned}
$$

Karatsuba's trick: Four for the product of three!

$$
\begin{aligned}
x &= 2^{n/2}x_1 + x_0 \\
y &= 2^{n/2}y_1 + y_0 \\
x \cdot y &= (2^{n/2}x_1 + x_0) \cdot (2^{n/2}y_1 + y_0) \\
&= 2^n(x_1 \cdot y_1) + 2^{n/2}((x_0 \cdot y_1) + (x_1 \cdot y_0)) + (x_0 \cdot y_0)
\end{aligned}
$$

Karatsuba's trick: Four for the product of three!

$$\begin{aligned} x &= 2^{n/2}x_1 + x_0 \\ y &= 2^{n/2}y_1 + y_0 \\ x \cdot y &= (2^{n/2}x_1 + x_0) \cdot (2^{n/2}y_1 + y_0) \\ &= 2^n(x_1 \cdot y_1) + 2^{n/2}((x_0 \cdot y_1) + (x_1 \cdot y_0)) + (x_0 \cdot y_0) \end{aligned}$$

Karatsuba's trick: Four for the product of three!

$$x = 2^{n/2} x_1 + x_0$$

$$y = 2^{n/2} y_1 + y_0$$

$$x \cdot y = (2^{n/2} x_1 + x_0) \cdot (2^{n/2} y_1 + y_0)$$

$$= 2^n (x_1 \cdot y_1) + 2^{n/2} ((x_0 \cdot y_1) + (x_1 \cdot y_0)) + (x_0 \cdot y_0)$$

$$t_1 = x_1 y_1, \quad t_0 = x_0 y_0, \quad t_{10} = (x_1 + x_0) \cdot (y_1 + y_0)$$

Karatsuba's trick: Four for the product of three!

$$x = 2^{n/2}x_1 + x_0$$

$$y = 2^{n/2}y_1 + y_0$$

$$x \cdot y = (2^{n/2}x_1 + x_0) \cdot (2^{n/2}y_1 + y_0)$$

$$= 2^n(x_1 \cdot y_1) + 2^{n/2}((x_0 \cdot y_1) + (x_1 \cdot y_0)) + (x_0 \cdot y_0)$$

$$t_1 = x_1y_1, \qquad t_0 = x_0y_0, \qquad t_{10} = (x_1 + x_0) \cdot (y_1 + y_0)$$

$$x_0y_1 + x_1y_0 = (x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0$$

Karatsuba's trick: Four for the product of three!

$$x = 2^{n/2}x_1 + x_0$$

$$y = 2^{n/2}y_1 + y_0$$

$$x \cdot y = (2^{n/2}x_1 + x_0) \cdot (2^{n/2}y_1 + y_0)$$

$$= 2^n(x_1 \cdot y_1) + 2^{n/2}((x_0 \cdot y_1) + (x_1 \cdot y_0)) + (x_0 \cdot y_0)$$

$$t_1 = x_1y_1, \qquad t_0 = x_0y_0, \qquad t_{10} = (x_1 + x_0) \cdot (y_1 + y_0)$$

$$\begin{aligned} x_0y_1 + x_1y_0 &= (x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0 \\ &= t_{10} - t_1 - t_0 \end{aligned}$$

Karatsuba's trick: Four for the product of three!

$$x = 2^{n/2} x_1 + x_0$$

$$y = 2^{n/2} y_1 + y_0$$

$$x \cdot y = (2^{n/2} x_1 + x_0) \cdot (2^{n/2} y_1 + y_0)$$

$$= 2^n (x_1 \cdot y_1) + 2^{n/2} ((x_0 \cdot y_1) + (x_1 \cdot y_0)) + (x_0 \cdot y_0)$$

$$t_1 = x_1 y_1, \qquad t_0 = x_0 y_0, \qquad t_{10} = (x_1 + x_0) \cdot (y_1 + y_0)$$

$$x_0 y_1 + x_1 y_0 = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0$$

$$= t_{10} - t_1 - t_0$$

All four multiplications, for only three multiplications!

Karatsuba's algorithm

KARATSUBA-MULTIPLY(x, y, n)

1. IF(n=1): RETURN $x \times y$.
2. ELSE:
   (a) $m = \lceil n/2 \rceil$. Set $x = 2^m x_1 + x_0$.
   (b) Set $y = 2^m y_1 + y_0$.

Karatsuba's algorithm

KARATSUBA-MULTIPLY(x, y, n)

number of bits

1. IF(n=1): RETURN $x \times y$.
2. ELSE:
   (a) $m = \lceil n/2 \rceil$. Set $x = 2^m x_1 + x_0$.
   (b) Set $y = 2^m y_1 + y_0$.

Karatsuba's algorithm

KARATSUBA-MULTIPLY(x, y, n)

1. IF(n=1): RETURN $x \times y$.
2. ELSE:
   (a) $m = \lceil n/2 \rceil$. Set $x = 2^m x_1 + x_0$.
   (b) Set $y = 2^m y_1 + y_0$.

Left-half of x

Karatsuba's algorithm

KARATSUBA-MULTIPLY(x, y, n)

1. IF(n=1): RETURN $x \times y$.
2. ELSE:
   (a) $m = \lceil n/2 \rceil$. Set $x = 2^m x_1 + x_0$.
   (b) Set $y = 2^m y_1 + y_0$.

Right-half of x

Karatsuba's algorithm

KARATSUBA-MULTIPLY(x, y, n)

1. IF(n=1): RETURN $x \times y$.
2. ELSE:
   (a) $m = \lceil n/2 \rceil$. Set $x = 2^m x_1 + x_0$.
   (b) Set $y = 2^m y_1 + y_0$.
   (c) $t_1 =$ KARATSUBA-MULTIPLY(x_1, y_1, m).

Karatsuba's algorithm

KARATSUBA-MULTIPLY(x, y, n)

1. IF(n=1): RETURN $x \times y$.
2. ELSE:
   (a) $m = \lceil n/2 \rceil$. Set $x = 2^m x_1 + x_0$.
   (b) Set $y = 2^m y_1 + y_0$.
   (c) $t_1 =$ KARATSUBA-MULTIPLY(x_1, y_1, m).
   (d) $t_0 =$ KARATSUBA-MULTIPLY(x_0, y_0, m).

Karatsuba's algorithm

KARATSUBA-MULTIPLY(x, y, n)

1. IF(n=1): RETURN $x \times y$.
2. ELSE:
   (a) $m = \lceil n/2 \rceil$. Set $x = 2^m x_1 + x_0$.
   (b) Set $y = 2^m y_1 + y_0$.
   (c) $t_1 =$ KARATSUBA-MULTIPLY(x_1, y_1, m).
   (d) $t_0 =$ KARATSUBA-MULTIPLY(x_0, y_0, m).
   (e) $t_{10} =$ KARATSUBA-MULTIPLY$(x_1 + x_0, y_1 + y_0, m)$.

Karatsuba's algorithm

Karatsuba-Multiply(x, y, n)

1. If(n=1): Return $x \times y$.
2. Else:
   (a) $m = \lceil n/2 \rceil$. Set $x = 2^m x_1 + x_0$.
   (b) Set $y = 2^m y_1 + y_0$.
   (c) $t_1 =$ Karatsuba-Multiply(x_1, y_1, m).
   (d) $t_0 =$ Karatsuba-Multiply(x_0, y_0, m).
   (e) $t_{10} =$ Karatsuba-Multiply$(x_1 + x_0, y_1 + y_0, m)$.
   (f) Return $2^{2m} t_1 + 2^m (t_{10} - t_1 - t_0) + t_0$

Karatsuba multiplication analysis

Proposition. Karatsuba's algorithm requires $O(n^{1.585})$ bit operations to multiply two n-bit integers.

Karatsuba multiplication analysis

Proposition. Karatsuba's algorithm requires $O(n^{1.585})$ bit operations to multiply two n-bit integers.

$$\boxed{T(n) = \underbrace{3T(\lceil n/2 \rceil)}_{\text{four recursive calls}} + \underbrace{O(n)}_{\text{9 adds/substractions}}}$$

Karatsuba multiplication analysis

Proposition. Karatsuba's algorithm requires $O(n^{1.585})$ bit operations to multiply two n-bit integers.

$$T(n) = \underbrace{3T(\lceil n/2 \rceil)}_{\text{four recursive calls}} + \underbrace{O(n)}_{\text{9 adds/substractions}}$$

Pf. Apply first-case of the master theorem to the recurrence: a = 3, b = 2, f(n) = O(n).

Karatsuba multiplication analysis

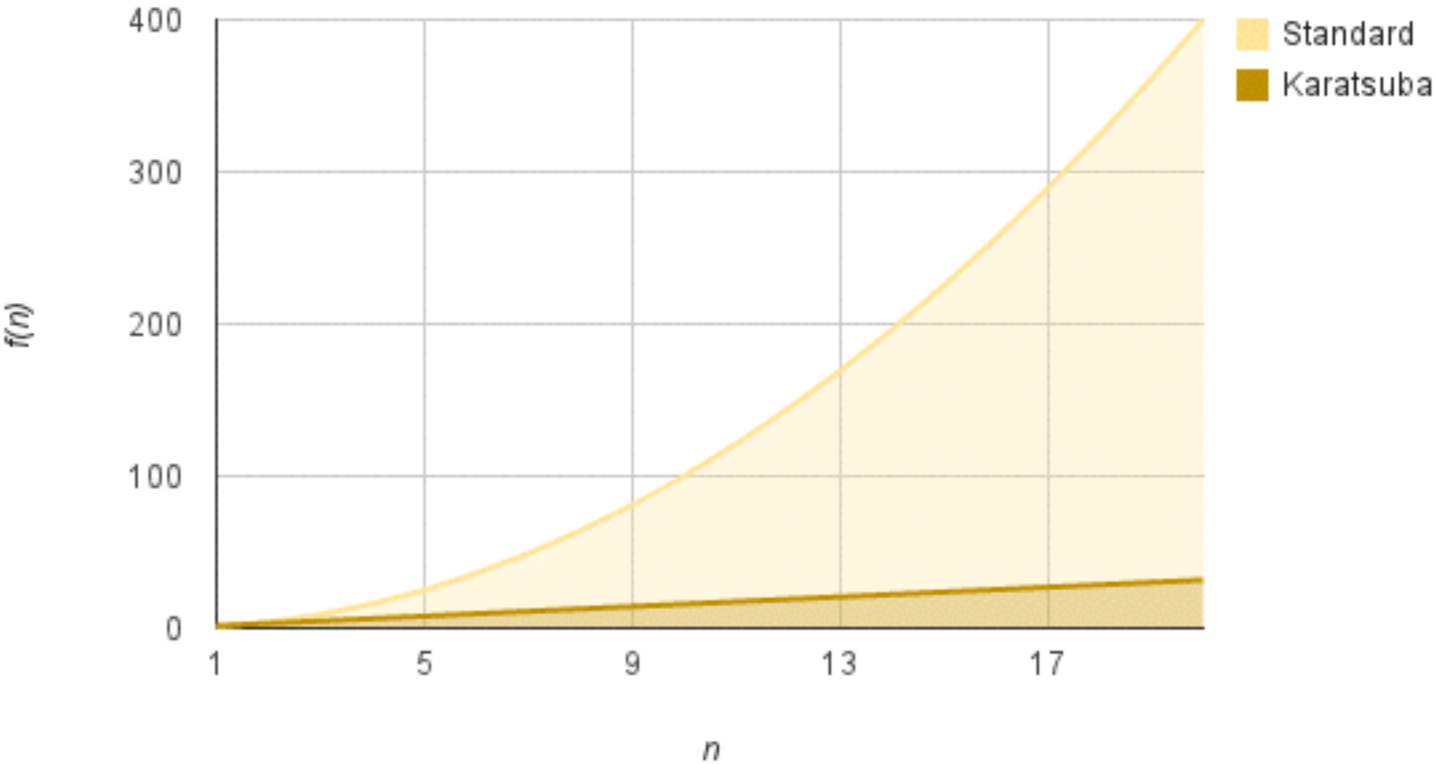
Proposition. Karatsuba's algorithm requires $O(n^{1.585})$ bit operations to multiply two n-bit integers.

$$T(n) = \underbrace{3T(\lceil n/2 \rceil)}_{\text{four recursive calls}} + \underbrace{O(n)}_{\text{9 adds/substractions}}$$

Pf. Apply first-case of the master theorem to the recurrence: a = 3, b = 2, f(n) = O(n).

Practice. Faster than grade-school algorithm for about 320-640 bits.

Karatsuba multiplication analysis



n^2 grows much quicker than $n^{1.585}$!

History of integer multiplication

year	algorithm	order of growth
?	brute force	$\Theta(n^2)$
1962	Karatsuba-Ofman	$\Theta(n^{1.585})$

History of integer multiplication

year	algorithm	order of growth
?	brute force	$\Theta(n^2)$
1962	Karatsuba-Ofman	$\Theta(n^{1.585})$
1963	Toom-3, Toom-4	$\Theta(n^{1.465})$, $\Theta(n^{1.404})$

History of integer multiplication

year	algorithm	order of growth
?	brute force	$\Theta(n^2)$
1962	Karatsuba-Ofman	$\Theta(n^{1.585})$
1963	Toom-3, Toom-4	$\Theta(n^{1.465})$, $\Theta(n^{1.404})$
1966	Toom-Cook	$\Theta(n^{1 + \varepsilon})$

History of integer multiplication

year	algorithm	order of growth
?	brute force	$\Theta(n^2)$
1962	Karatsuba-Ofman	$\Theta(n^{1.585})$
1963	Toom-3, Toom-4	$\Theta(n^{1.465})$, $\Theta(n^{1.404})$
1966	Toom-Cook	$\Theta(n^{1+\varepsilon})$
1971	Schönhage–Strassen	$\Theta(n \log n \log \log n)$

History of integer multiplication

year	algorithm	order of growth
?	brute force	$\Theta(n^2)$
1962	Karatsuba-Ofman	$\Theta(n^{1.585})$
1963	Toom-3, Toom-4	$\Theta(n^{1.465})$, $\Theta(n^{1.404})$
1966	Toom-Cook	$\Theta(n^{1+\varepsilon})$
1971	Schönhage–Strassen	$\Theta(n \log n \log \log n)$
2007	Fürer	$n \log n \, 2^{O(\log^* n)}$

History of integer multiplication

year	algorithm	order of growth
?	brute force	$\Theta(n^2)$
1962	Karatsuba-Ofman	$\Theta(n^{1.585})$
1963	Toom-3, Toom-4	$\Theta(n^{1.465})$, $\Theta(n^{1.404})$
1966	Toom-Cook	$\Theta(n^{1+\varepsilon})$
1971	Schönhage–Strassen	$\Theta(n \log n \log \log n)$
2007	Fürer	$n \log n \, 2^{O(\log^* n)}$
?	?	$\Theta(n \log n)$

History of integer multiplication

year	algorithm	order of growth
?	brute force	$\Theta(n^2)$
1962	Karatsuba-Ofman	$\Theta(n^{1.585})$
1963	Toom-3, Toom-4	$\Theta(n^{1.465})$, $\Theta(n^{1.404})$
1966	Toom-Cook	$\Theta(n^{1+\varepsilon})$
1971	Schönhage–Strassen	$\Theta(n \log n \log \log n)$
2007	Fürer	$n \log n \, 2^{O(\log^* n)}$
?	?	$\Theta(n \log n)$

GNU Multiple Precision Library uses one of five different algorithm depending on size of operands.

GMP
«Arithmetic without limitations»

History of integer multiplication

year	algorithm	order of growth
?	brute force	$\Theta(n^2)$
1962	Karatsuba-Ofman	$\Theta(n^{1.585})$
1963	Toom-3, Toom-4	$\Theta(n^{1.465})$, $\Theta(n^{1.404})$
1966	Toom-Cook	$\Theta(n^{1+\varepsilon})$
1971	Schönhage–Strassen	$\Theta(n \log n \log \log n)$
2007	Fürer	$n \log n \, 2^{O(\log^* n)}$
?	?	$\Theta(n \log n)$

GNU Multiple Precision Library uses one of five different algorithm depending on size of operands.

Used in maple, mathematica, matlab, crypto, …

GMP
«Arithmetic without limitations»

Plan for Today

Master theorem

Integer multiplication

Exponentiation

Plan for Today

Master theorem

Integer multiplication

Exponentiation : Very useful in number theory,
Cryptography

Basic arithmetic ops

Exponentiation
INPUT: Given two numbers a, n
OUTPUT: a^n in binary format.

Example: a = 11(3), n = 10.
1110011010101001

Absolutely critical in cryptography.

Basic arithmetic ops

Exponentiation
INPUT: Given two numbers a, n
OUTPUT: a^n in binary format.

NAIVE-EXPONENTIATE(a,n)

Basic arithmetic ops

Exponentiation
INPUT: Given two numbers a, n
OUTPUT: a^n in binary format.

NAIVE-EXPONENTIATE(a,n)

1. SET A = 1.
2. FOR $i = 1 : n$,
   SET $A = a \cdot A$.
3. RETURN A.

Basic arithmetic ops

Exponentiation
INPUT: Given two numbers a, n
OUTPUT: a^n in binary format.

NAIVE-EXPONENTIATE(a,n)

1. SET A = 1.
2. FOR $i = 1 : n$,
   SET $A = a \cdot A$.
3. RETURN A.

Basic arithmetic ops

Exponentiation
INPUT: Given two numbers a, n
OUTPUT: a^n in binary format.

NAIVE-EXPONENTIATE(a,n)

1. SET A = 1.
2. FOR $i = 1 : n$,
   SET $A = a \cdot A$.
3. RETURN A.

What is the running-time?

Basic arithmetic ops

Exponentiation
INPUT: Given two numbers a, n
OUTPUT: a^n in binary format.

NAIVE-EXPONENTIATE(a,n)

1. SET A = 1.
2. FOR $i = 1 : n$,
   SET $A = a \cdot A$.
3. RETURN A.

What is the running-time?

Let us specialize to a = 3.

Exponentiation

Exponentiation
INPUT: Given n
OUTPUT: 3^n in binary format.

Exponentiation

Exponentiation
INPUT: Given n
OUTPUT: 3^n in binary format.

n=1: 3^1 = 11
n=2: 3^2 = 1001
n=3: 3^3 = 11011
n=4: 3^4 = 1010001
n=5: 3^5 = 11110011
n=6: 3^6 = ...

Exponentiation

Exponentiation
INPUT: Given n
OUTPUT: 3^n in binary format.

NAIVE-EXPONENTIATE(3,N)

1. SET $A_0 = 1$.
2. FOR $i = 1 : n$,
   SET $A_i = 3 \cdot A_{i-1}$.
3. RETURN A_n.

What is the running-time?

Exponentiation

Exponentiation
INPUT: Given n
OUTPUT: 3^n in binary format.

NAIVE-EXPONENTIATE(3,N)

1. SET $A_0 = 1$.
2. FOR $i = 1 : n$,
   SET $A_i = 3 \cdot A_{i-1}$.
3. RETURN A_n.

O(1)

What is the running-time?

Exponentiation: Running-time

Exponentiation
INPUT: Given n
OUTPUT: 3^n in binary format.

NAIVE-EXPONENTIATE(3,N)

1. SET $A_0 = 1$.
2. FOR $i = 1 : n$,
   SET $A_i = 3 \cdot A_{i-1}$.
3. RETURN A_n.

Time in i'th iteration?

What is the running-time?

Exponentiation: Running-time

Exponentiation
INPUT: Given n
OUTPUT: 3^n in binary format.

NAIVE-EXPONENTIATE(3,N)

1. SET $A_0 = 1$.
2. FOR $i = 1 : n$,
   SET $A_i = 3 \cdot A_{i-1}$.
3. RETURN A_n.

O(# bits in A_{i-1})

What is the running-time?

Exponentiation: Running-time

Exponentiation
INPUT: Given n
OUTPUT: 3^n in binary format.

NAIVE-EXPONENTIATE(3,N)

1. SET $A_0 = 1$.
2. FOR $i = 1 : n$,
   SET $A_i = 3 \cdot A_{i-1}$.
3. RETURN A_n.

O(# bits in A_{i-1}) = O(i).

What is the running-time?

Exponentiation: Running-time

Running time = $$\begin{aligned} &O(1) + O(1 + 2 + \cdots + n) \\ &= O(1) + O(n(n+1)/2) \\ &= O(n^2). \end{aligned}$$

$\textsc{Naive-Exponentiate}(3,\textsc{n})$

1. $\textsc{Set}\ A_0 = 1.$
2. $\textsc{For}\ i = 1 : n,$
   $\textsc{Set}\ A_i = 3 \cdot A_{i-1}.$
3. $\textsc{Return}\ A_n.$

O(# bits in A_{i-1})
= O(i).

What is the running-time?

Exponentiation: Running-time

Running time = $$\begin{aligned} &O(1) + O(1 + 2 + \cdots + n) \\ &= O(1) + O(n(n+1)/2) \\ &= O(n^2). \end{aligned}$$

Proposition: Naive-Exponentiate runs in $O(n^2)$ time.

NAIVE-EXPONENTIATE(3,N)

1. SET $A_0 = 1$.
2. FOR $i = 1 : n$,
   SET $A_i = 3 \cdot A_{i-1}$.
3. RETURN A_n.

Exponentiation: Running-time

Running time = $$\begin{aligned} & O(1) + O(1 + 2 + \cdots + n) \\ &= O(1) + O(n(n+1)/2) \\ &= O(n^2). \end{aligned}$$

Proposition: Naive-Exponentiate runs in $O(n^2)$ time.

Can we do better?

NAIVE-EXPONENTIATE(3,N)

1. SET $A_0 = 1$.
2. FOR $i = 1 : n$,
   SET $A_i = 3 \cdot A_{i-1}$.
3. RETURN A_n.

Exponentiation: Running-time

Running time = $$\begin{aligned} &O(1) + O(1 + 2 + \cdots + n) \\ &= O(1) + O(n(n+1)/2) \\ &= O(n^2). \end{aligned}$$

Proposition: Naive-Exponentiate runs in $O(n^2)$ time.

Can we do better?

YES WE CAN!

Fast exponentiation

Exponentiation
INPUT: Given n
OUTPUT: 3^n in binary format.

Recursive view of algorithm:

NAIVE-EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   $A_{n-1} =$ NAIVE-EXPONENTIATE(3,N-1)
   RETURN $3 \cdot A_{n-1}$.

Fast exponentiation

Exponentiation
INPUT: Given n
OUTPUT: 3^n in binary format.

Recursive view of algorithm:

NAIVE-EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   $A_{n-1} =$ NAIVE-EXPONENTIATE(3,N-1)
   RETURN $3 \cdot A_{n-1}$.

Not dividing enough!

Fast exponentiation

Divide and conquer algorithm:

EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE

Fast exponentiation

Divide and conquer algorithm:

$\textsc{Exponentiate}(3,\textsc{n})$

1. $\textsc{If}$ $n = 1$, $\textsc{Return}$ 3.
2. $\textsc{Else}$
   (a) $\textsc{Set}$ $A_\ell = \textsc{Exponentiate}(3, \lfloor n/2 \rfloor)$.
   (b) $\textsc{Set}$ $A_r = \textsc{Exponentiate}(3, \lceil n/2 \rceil)$.

Fast exponentiation

Divide and conquer algorithm:

$\textsc{Exponentiate}(3,\textsc{n})$

1. $\textsc{If}\ n = 1,\ \textsc{Return}\ 3.$
2. $\textsc{Else}$
   (a) $\textsc{Set}\ A_\ell = \textsc{Exponentiate}(3, \lfloor n/2 \rfloor).$

   (b) $\textsc{Set}\ A_r = \textsc{Exponentiate}(3, \lceil n/2 \rceil).$

   (c) $\textsc{Return}\ A_\ell \cdot A_r.$

Fast exponentiation

Divide and conquer algorithm:

EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   (a) SET $A_\ell =$ EXPONENTIATE$(3, \lfloor n/2 \rfloor)$.
   (b) SET $A_r =$ EXPONEN
   (c) RETURN $A_\ell \cdot A_r$.

How to multiply?

Fast exponentiation

Divide and conquer algorithm:

$\textsc{Exponentiate}(3,\textsc{n})$

1. $\textsc{If}$ $n = 1$, $\textsc{Return}$ 3.
2. $\textsc{Else}$

   (a) $\textsc{Set}$ $A_\ell = \textsc{Exponentiate}(3, \lfloor n/2 \rfloor)$.

   (b) $\textsc{Set}$ $A_r = \textsc{Exponentiate}(3, \lceil n/2 \rceil)$.

   (c) $\textsc{Return}$ $\textsc{Karatsuba-Multiply}(A_\ell, A_r)$.

Fast exponentiation

Divide and conquer algorithm:

EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   (a) SET A_ℓ = EXPONENTIATE($3, \lfloor n/2 \rfloor$).
   (b) SET A_r = EXPONENTIATE($3, \lceil n/2 \rceil$).
   (c) RETURN KARATSUBA-MULTIPLY(A_ℓ, A_r).

What is the running time?

Fast exponentiation

Divide and conquer algorithm:

$\textsc{Exponentiate}(3,\textsc{n})$

1. $\textsc{If}$ $n = 1$, $\textsc{Return}$ 3.
2. $\textsc{Else}$
   (a) $\textsc{Set}$ $A_\ell = \textsc{Exponentiate}(3, \lfloor n/2 \rfloor)$.
   (b) $\textsc{Set}$ $A_r = \textsc{Exponentiate}(3, \lceil n/2 \rceil)$.
   (c) $\textsc{Return}$ $\textsc{Karatsuba-Multiply}(A_\ell, A_r)$.

Is this the best or can we do better?

Fast exponentiation

Divide and conquer algorithm:

$\textsc{Exponentiate}(3,\textsc{n})$

1. $\textsc{If}\ n = 1,\ \textsc{Return}\ 3.$
2. $\textsc{Else}$
   (a) $\textsc{Set}\ A_\ell = \textsc{Exponentiate}(3, \lfloor n/2 \rfloor).$
   (b) $\textsc{Set}\ A_r = \textsc{Exponentiate}(3, \lceil n/2 \rceil).$
   (c) $\textsc{Return Karatsuba-Multiply}(A_\ell, A_r).$

Do you really need two calls?

Is this the best or can we do better?

Fast exponentiation

Divide and conquer algorithm:

EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   (a) SET $A_\ell =$ EXPONENTIATE$(3, \lfloor n/2 \rfloor)$.
   (b) SET $A_r =$ EXPONENTIATE$(3, \lceil n/2 \rceil)$.
   (c) RETURN KARATSUBA-MULTIPLY(A_ℓ, A_r).

Ex: If n even? NO!

Is this the best or can we do better?

Fast exponentiation

Divide and conquer algorithm:

$\textsc{Exponentiate}(3,\textsc{n})$

1. $\textsc{If}\ n = 1,\ \textsc{Return}\ 3.$
2. $\textsc{Else}$
   (a) $\textsc{Set}\ A_\ell = \textsc{Exponentiate}(3, \lfloor n/2 \rfloor).$
   (b) $\textsc{Set}\ A_r = \textsc{Exponentiate}(3, \lceil n/2 \rceil).$
   (c) $\textsc{Return}\ \textsc{Karatsuba-Multiply}(A_\ell, A_r).$

Ex: If n even? NO!
If n odd? No!

Is this the best or can we do better?

Fast exponentiation

Divide and conquer algorithm:

Fast-Exponentiate(3,n)

1. If $n = 1$, Return 3.
2. Else
   (a) Set A_ℓ = Exponentiate$(3, \lfloor n/2 \rfloor)$.
   (b) If n Even
   (c)

Fast exponentiation

Divide and conquer algorithm:

FAST-EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   (a) SET A_ℓ = EXPONENTIATE($3, \lfloor n/2 \rfloor$).
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD

Fast exponentiation

Divide and conquer algorithm:

$\textsc{Fast-Exponentiate}(3,\textsc{n})$

1. $\textsc{If}\ n = 1,\ \textsc{Return}\ 3.$
2. $\textsc{Else}$
   (a) $\textsc{Set}\ A_\ell = \textsc{Exponentiate}(3, \lfloor n/2 \rfloor).$
   (b) $\textsc{If}\ n\ \textsc{Even}$
   $\textsc{Return}\quad \textsc{Karatsuba-Multiply}(A_\ell, A_\ell).$
   (c) $\textsc{If}\ n\ \textsc{Odd}$
   $\textsc{Return}\quad 3 \cdot \textsc{Karatsuba-Multiply}(A_\ell, A_\ell).$

What is the running time?

Fast exponentiation: Running-time

T(n) = Time taken on input n.

FAST-EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   (a) SET A_ℓ = EXPONENTIATE($3, \lfloor n/2 \rfloor$).
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN $3\cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

O(1)

Fast exponentiation: Running-time

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

FAST-EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE

T(n/2)

   (a) SET A_ℓ = EXPONENTIATE$(3, \lfloor n/2 \rfloor)$.
   (b) IF n EVEN
       RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
       RETURN $3\cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

Fast exponentiation: Running-time

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

FAST-EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   (a) SET $A_\ell =$ EXPONENTIATE(3, $\lfloor n/2 \rfloor$).
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN $3\cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

$O((\#\text{ bits in } A_\ell)^{(\log_2 3)})$

Fast exponentiation: Running-time

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

FAST-EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   (a) SET $A_\ell =$ EXPONENTIATE , $\lfloor n/2 \rfloor)$.
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN $3\cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

$$O(n^{(\log_2 3)})$$

Fast exponentiation: Running-time

$$T(n) = O(1) + T(n/2) + O(n^{\log_2 3}) +$$

FAST-EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   (a) SET A_ℓ = EXPONENTIATE(3, $\lfloor n/2 \rfloor$).
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN 3·KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

O(n)

Fast exponentiation: Running-time

$$T(n) = O(1) + T(n/2) + O(n^{\log_2 3}) + O(n)$$

FAST-EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   (a) SET A_ℓ = EXPONENTIATE(3, $\lfloor n/2 \rfloor$).
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN $3\cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

O(n)

Fast exponentiation: Running-time

$$T(n) = T(n/2) + O(n^{\log_2 3})$$

$\textsc{Fast-Exponentiate}(3,\textsc{n})$

1. $\textsc{If}$ $n = 1$, $\textsc{Return}$ 3.
2. $\textsc{Else}$
   (a) $\textsc{Set}$ $A_\ell = \textsc{Exponentiate}(3, \lfloor n/2 \rfloor)$.
   (b) $\textsc{If}$ n $\textsc{Even}$
   $\textsc{Return}$ $\textsc{Karatsuba-Multiply}(A_\ell, A_\ell)$.
   (c) $\textsc{If}$ n $\textsc{Odd}$
   $\textsc{Return}$ $3 \cdot \textsc{Karatsuba-Multiply}(A_\ell, A_\ell)$.

Fast exponentiation: Running-time

$$T(n) = T(n/2) + O(n^{\log_2 3})$$

Proposition: Fast-Exponentiate runs in $O(n^{1.585})$ time.

FAST-EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   (a) SET A_ℓ = EXPONENTIATE(3, $\lfloor n/2 \rfloor$).
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN 3·KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

Fast exponentiation: Running-time

$$T(n) = T(n/2) + O(n^{\log_2 3})$$

Proposition: Fast-Exponentiate runs in $O(n^{1.585})$ time.

Proof: Apply Master theorem,

FAST-EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   (a) SET $A_\ell =$ EXPONENTIATE$(3, \lfloor n/2 \rfloor)$.
   (b) IF n EVEN
       RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
       RETURN $3\cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

Fast exponentiation: Running-time

$$T(n) = T(n/2) + O(n^{\log_2 3})$$

Proposition: Fast-Exponentiate runs in $O(n^{1.585})$ time.

Proof: Apply Master theorem, case 3 to T.
a = 1, b = 2, f(n) = $O(n^{1.585})$.
Big improvement over quadratic!

FAST-EXPONENTIATE(3,N)

1. IF $n = 1$, RETURN 3.
2. ELSE
   (a) SET A_ℓ = EXPONENTIATE(3, $\lfloor n/2 \rfloor$).
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN 3·KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

Summary for today

Integer multiplication

Fast exponentiation

Summary for today

Integer multiplication

Fast exponentiation

When in doubt,
Divide it up!

General case?

Exponentiation
INPUT: Given two numbers a, n
OUTPUT: a^n in binary format.

Example: a = 101011, n = 10?

Absolutely critical in cryptography.

General case?

Exponentiation
INPUT: Given two numbers a, n
OUTPUT: a^n in binary format.

Example: a = 101011, n = 10?

Absolutely critical in cryptography.

Same algorithm!

Fast exponentiation

Divide and conquer algorithm:

FAST-EXPONENTIATE(a, n)

1. IF $n = 1$, RETURN a.
2. ELSE
   (a) SET $A_\ell = $ EXPONENTIATE$(a, \lfloor n/2 \rfloor)$.
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN $a \cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

What is the running time?

Fast exponentiation

T(n) = Run-time on n.
m = # bits in a.

$\textsc{Fast-Exponentiate}(a, n)$

1. $\textsc{If}$ $n = 1$, $\textsc{Return}$ a.
2. $\textsc{Else}$
   (a) $\textsc{Set}$ $A_\ell = \textsc{Exponentiate}(a, \lfloor n/2 \rfloor)$.
   (b) $\textsc{If}$ n $\textsc{Even}$
   $\textsc{Return}$ $\textsc{Karatsuba-Multiply}(A_\ell, A_\ell)$.
   (c) $\textsc{If}$ n $\textsc{Odd}$
   $\textsc{Return}$ $a \cdot \textsc{Karatsuba-Multiply}(A_\ell, A_\ell)$.

What is the running time?

Fast exponentiation

T(n) = Run-time on n.
m = # bits in a.

$\textsc{Fast-Exponentiate}(a, n)$

1. $\textsc{If}\ n = 1,\ \textsc{Return}\ a.$

O(1)

2. $\textsc{Else}$
   (a) $\textsc{Set}\ A_\ell = \textsc{Exponentiate}(a, \lfloor n/2 \rfloor).$
   (b) $\textsc{If}\ n\ \textsc{Even}$
       $\textsc{Return}\quad \textsc{Karatsuba-Multiply}(A_\ell, A_\ell).$
   (c) $\textsc{If}\ n\ \textsc{Odd}$
       $\textsc{Return}\quad a \cdot \textsc{Karatsuba-Multiply}(A_\ell, A_\ell).$

What is the running time?

Fast exponentiation

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

$\textsc{Fast-Exponentiate}(a, n)$

1. $\textsc{If}$ $n = 1$, $\textsc{Return}$ a.
2. $\textsc{Else}$

T(n/2)

   (a) $\textsc{Set}$ $A_\ell = \textsc{Exponentiate}(a, \lfloor n/2 \rfloor)$.

   (b) $\textsc{If}$ n $\textsc{Even}$
   $\textsc{Return}$ $\textsc{Karatsuba-Multiply}(A_\ell, A_\ell)$.

   (c) $\textsc{If}$ n $\textsc{Odd}$
   $\textsc{Return}$ $a \cdot \textsc{Karatsuba-Multiply}(A_\ell, A_\ell)$.

What is the running time?

Fast exponentiation

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

FAST-EXPONENTIATE(a, n)

1. IF $n = 1$, RETURN a.
2. ELSE
   (a) SET $A_\ell =$ EXPONENTIATE$(a, \lfloor n/2 \rfloor)$.
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN $a \cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

$O((\#\text{ bits in } A_\ell)^{(\log_2 3)})$

What is the running time?

Fast exponentiation

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

FAST-EXPONENTIATE(a, n)

1. IF $n = 1$, RETURN a.

\# bits < (# bits in a)·n

$O((\#\text{ bits in } A_\ell)^{(\log_2 3)})$

(a) SET $A_\ell =$ EXPONENTIATE(, $\lfloor n/2 \rfloor$).

(b) IF n EVEN
RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

(c) IF n ODD
RETURN $a\cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

What is the running time?

Fast exponentiation

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

FAST-EXPONENTIATE(a, n)

1. IF $n = 1$, RETURN a.

\# bits < m*n

$O((\# \text{ bits in } A_\ell)^{(\log_2 3)})$

   (a) SET A_ℓ = EXPONENTIATE$(a, \lfloor n/2 \rfloor)$.

   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

   (c) IF n ODD
   RETURN $a \cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

What is the running time?

Fast exponentiation

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

FAST-EXPONENTIATE(a, n)

1. IF $n = 1$, RETURN a.

$$O((mn)^{(\log_2 3)})$$

2. ELSE

   (a) SET $A_\ell =$ EXPONENTIATE$(a, \lfloor n/2 \rfloor)$.

   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

   (c) IF n ODD
   RETURN $a\cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

Fast exponentiation

$$T(n) = O(1) + T(n/2) + O((mn)^{\log_2 3})$$

FAST-EXPONENTIATE(a, n)

1. IF $n = 1$, RETURN a.
2. ELSE
   (a) SET $A_\ell =$ EXPONENTIATE$(a, \lfloor n/2 \rfloor)$.
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN $a \cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

$O((mn)^{(\log_2 3)})$

Fast exponentiation

$$T(n) = O(1) + T(n/2) + O((mn)^{\log_2 3})$$

$\textsc{Fast-Exponentiate}(a, n)$

1. $\textsc{If}$ $n = 1$, $\textsc{Return}$ a.
2. $\textsc{Else}$
   (a) $\textsc{Set}$ $A_\ell = \textsc{Exponentiate}(a, \lfloor n/2 \rfloor)$.
   (b) $\textsc{If}$ n $\textsc{Even}$
   $\textsc{Return}$ $\textsc{Karatsuba-Multiply}(A_\ell, A_\ell)$.
   (c) $\textsc{If}$ n $\textsc{Odd}$
   $\textsc{Return}$ $a \cdot \textsc{Karatsuba-Multiply}(A_\ell, A_\ell)$.

Fast exponentiation: Running-time

$$T(n) = T(n/2) + O((mn)^{\log_2 3})$$

Proposition: Fast-Exponentiate runs in $O((mn)^{1.585})$ time.

FAST-EXPONENTIATE(a, n)

1. IF $n = 1$, RETURN a.
2. ELSE
   (a) SET A_ℓ = EXPONENTIATE$(a, \lfloor n/2 \rfloor)$.
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN $a \cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

Fast exponentiation: Running-time

$$T(n) = T(n/2) + O((mn)^{\log_2 3})$$

Proposition: Fast-Exponentiate runs in $O((mn)^{1.585})$ time.

Proof: Apply Master theorem,

FAST-EXPONENTIATE(a, n)

1. IF $n = 1$, RETURN a.
2. ELSE
   (a) SET $A_\ell =$ EXPONENTIATE$(a, \lfloor n/2 \rfloor)$.
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN $a\cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).

Fast exponentiation: Running-time

$$T(n) = T(n/2) + O((mn)^{\log_2 3})$$

Proposition: Fast-Exponentiate runs in $O((mn)^{1.585})$ time.

Proof: Apply Master theorem, case 3 to T.
$a = 1, b = 2, f(n) = O((mn)^{1.585})$.
Big improvement over quadratic!

FAST-EXPONENTIATE(a, n)

1. IF $n = 1$, RETURN a.
2. ELSE
   (a) SET $A_\ell =$ EXPONENTIATE$(a, \lfloor n/2 \rfloor)$.
   (b) IF n EVEN
   RETURN KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).
   (c) IF n ODD
   RETURN $a\cdot$KARATSUBA-MULTIPLY(A_ℓ, A_ℓ).