Divide-and-Conquer Algorithms Analysis

Divide-and-Conquer algorithms

Previously:

Divide-and-Conquer as an algorithm development technique.

Today:

Analysis of Divide-and-Conquer algorithms in terms of correctness and (especially) running time.

Divide-and-Conquer algorithms, recap

The same as recursive algorithms.

A general algorithm development method:

1. Divide the problem into smaller subproblems (of the same type).
2. Solve the subproblems using recursion (i.e. call the algorithm itself, but with the smaller inputs).
3. Construct a solution to the problem based on the solution of the subproblems.

Base case: Problems of smallest size are solved directly (without recursion).

This is a general algorithm development method, with many applications.

For each new algorithm, points 1 and 3 must be developed. Point 2 is always the same. The solution for the base case must also be developed, but is usually trivial.

General structure of Divide-and-Conquer code

If base case ($n = O(1)$):

- Work

If NOT base case:

- Work
- Recursive call
- Work
- Recursive call
- Work

(There don't always have to be two recursive calls. Some recursive algorithms have just one, and some have more than two).

Divide-and-Conquer examples

Mergesort:

- Divide the input into two parts X and Y (trivial).
- Sort each part separately (recursion).
- Merge the two sorted parts into one sorted part (real work).

Base case: $n \leq 1$ (trivial).

Quicksort:

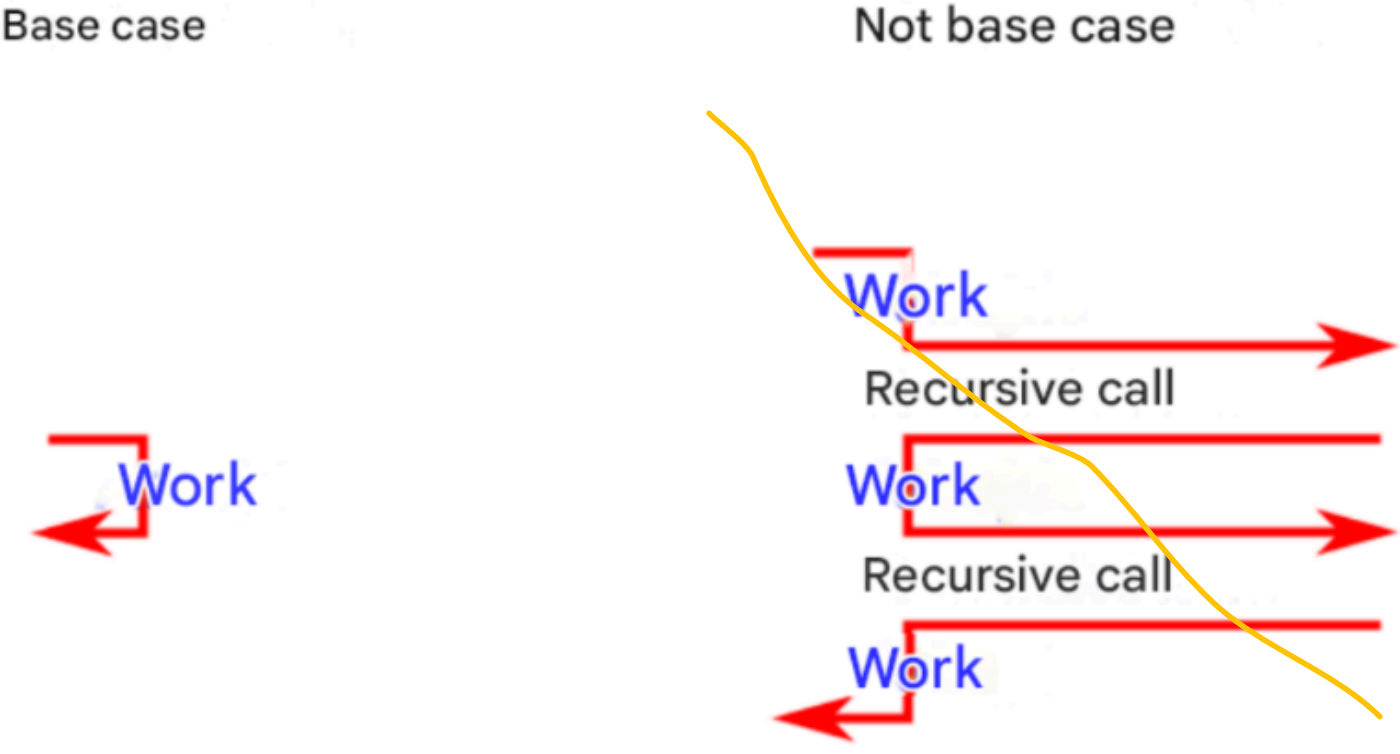
- Divide the input into two parts X and Y such that $X \leq Y$ (real work).
- Sort each part separately (recursion).
- Return X followed by Y (trivial)

Base case: $n \leq 1$ (trivial).

A third example is inorder traversal of a binary search tree: Two recursive calls (on the right and left subtrees) with O(1) work between them (printing the key in the current node).

What happens when Divide-and-Conquer algorithms execute?

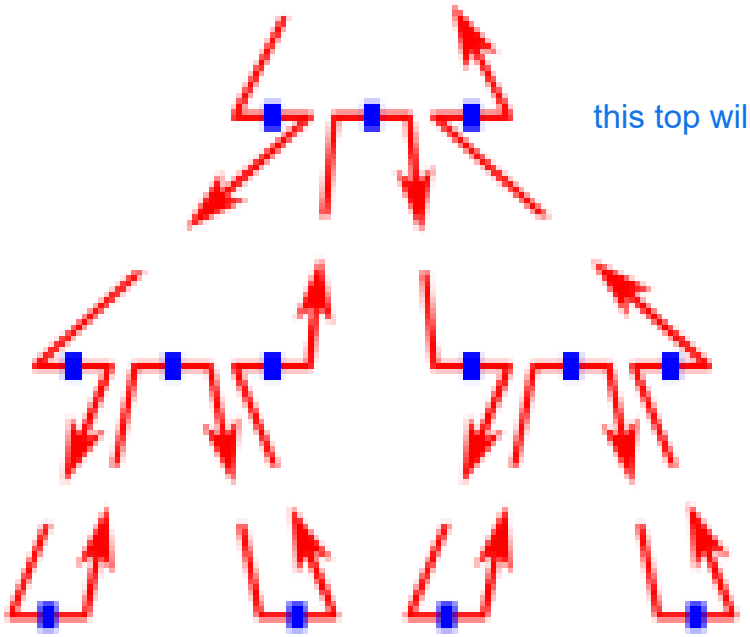
Flow of control



if we divide at the orange line, we will get next slide

What happens when Divide-and-Conquer algorithms execute?

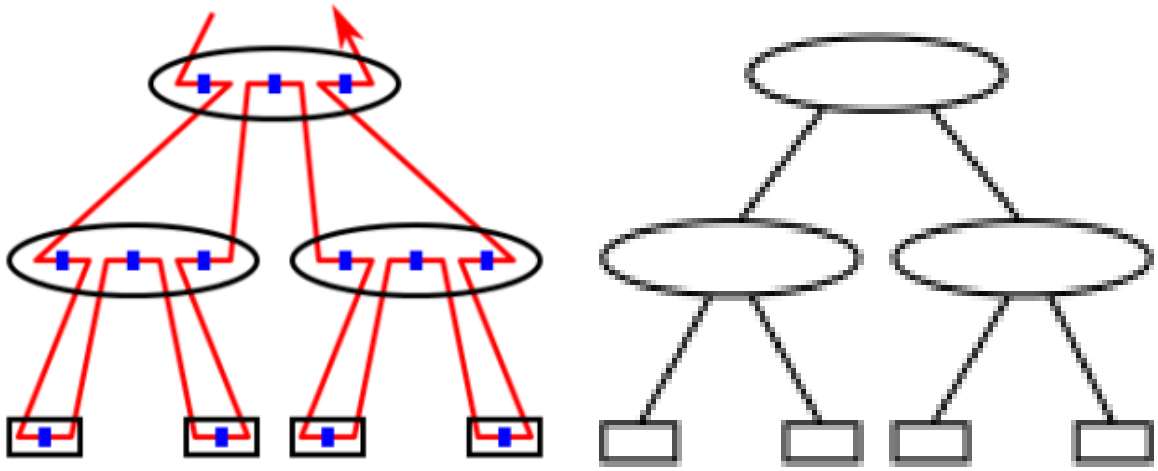
Global flow of control:



this top will be excatly what we saw before

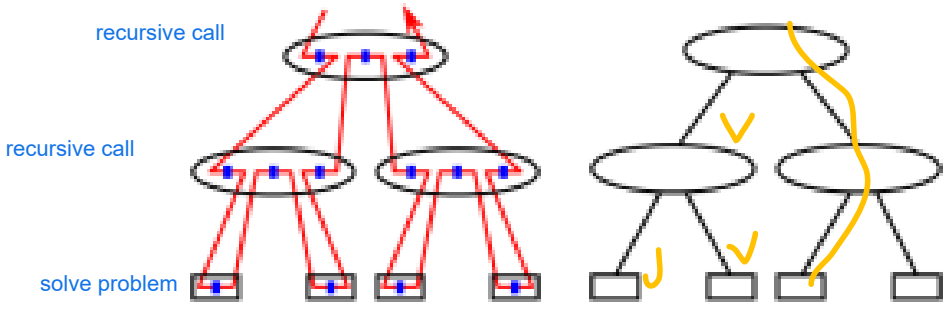
Recursion Trees

This structure for flow of control can be illustrated via trees, which have a node for each call of the code. The base cases are leaves, non-base cases are internal nodes, and the fanout of the tree (number of children of nodes) is equal to the number of recursive calls in the code.



These trees are called recursion trees.

What happens when Divide-and-Conquer algorithms execute?



everything on the path down to the call (square) is begin paused on the stack. Everything else is has ben executed or not ran yet

At any given time, the total execution (the red path) has reached some node v. = blue dot

At this time, the call corresponding to v is the one from which actions are being performed. All calls corresponding to nodes on the path from v to the root are paused.

The state of each of these calls (contents of their variables, which command they reached, etc.) is stored by the computer on a stack so that the execution of the calls is not mixed up. Calls corresponding to all other nodes in the tree are either completely finished or not started at all.

- Call of child in the recursion tree = push on stack.
- End of execution of a node = pop from stack.

Correctness of Divide-and-Conquer algorithms

Correctness of Divide-and-Conquer algorithms is argued from the bottom up in the recursion tree:

- Arguments that base case calls answer correctly (usually trivial).
- For a non-base case call, arguments that if the recursive calls answer correctly, then this, together with the local work, will make the answer correct in the current call.

That is, that correctness of calls with large inputs follows from correctness of calls with smaller inputs and the actions involved in constructing a solution for the large input from the solutions for the smaller inputs.

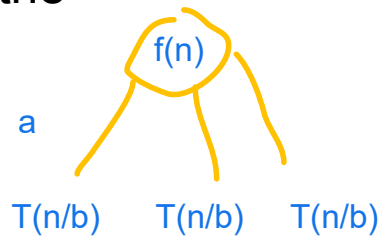
[Formally, one can also say that correctness is shown via induction on the input size. The base case for the recursion is the base case for the induction proof.]

The argument itself is individual for each algorithm. It is usually developed together with the algorithm (in practice, it is difficult to get the idea for the algorithm without having the idea of why it works).

Recursion equations

We will now look at the running times of recursive algorithms. We must first see how these can be described by recursion equations.
Let the worst-case running time on input of size n be T(n). If a recursive algorithm for problems of size n makes a recursive call, all of which are on subproblems of size n/b, and does Θ(f (n)) local work, then the running time must be $T(n)$:

$$T(n) = a \cdot T(n/b) + \Theta(f(n)) \quad if\ n > 1$$
$$= \Theta(1) \quad if\ n \leq 1$$



so the number of recursive calls multiplied by size of subproblem + local work

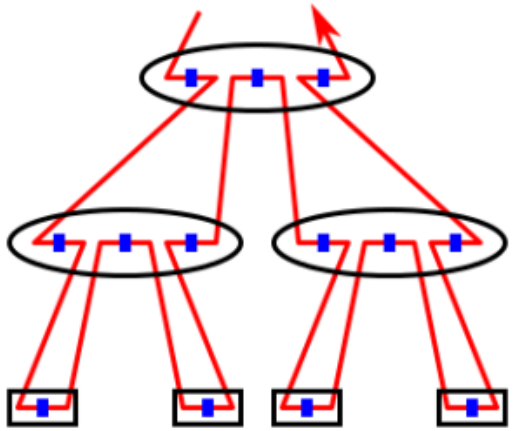
The last line is always the same and is therefore often omitted. It is also often implied that we use asymptotic notation (so we write **f(n)** instead of Θ(f (n)) and **1** instead of Θ(1)).

Example: Mergesort has two recursive calls of size n/2 and does Θ(n) local work. Its recursion equation is therefore written

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

Running time of Divide-and-Conquer algorithms

A recursion equation describes the structure of a recursive algorithm and provides a recursive description of its running time (i.e that T(n) is described in terms of T on a smaller input size).
But we naturally also want to find a direct expression (a function of n) for the algorithm's running time T(n).
The total work of a recursive algorithm is equal to the sum of local work in the algorithm's recursion tree:



That is, we want to find this sum.

The recursion tree method for calculating running time

For a recursive algorithm (or a recurrence relation), annotate the nodes in the recursion tree (for the algorithm or the equation) with:

- The input size for the call to the node.
- The resulting work done in this node.

Then, find the total sum of the work in all nodes as follows:

- First, determine the height of the tree (number of levels).
- Sum up the work for each level of the recursion tree separately.
- Sum the resulting values from all levels.

Examples follow.

Example1

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

$$a = 2$$

$$b = 2$$

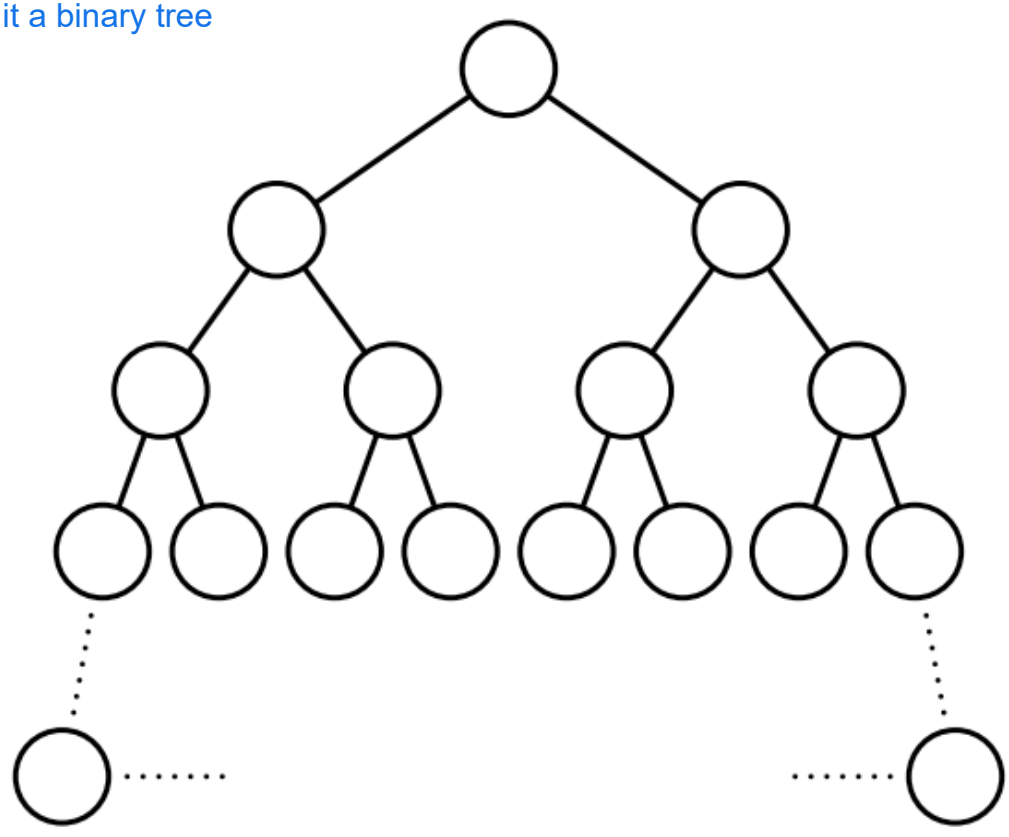
$$f(n) = n$$

Example1

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

$$a = 2,\ b = 2,\ f(x) = x$$

a makes it a binary tree



Draw a tree with $a = 2$.

Example1

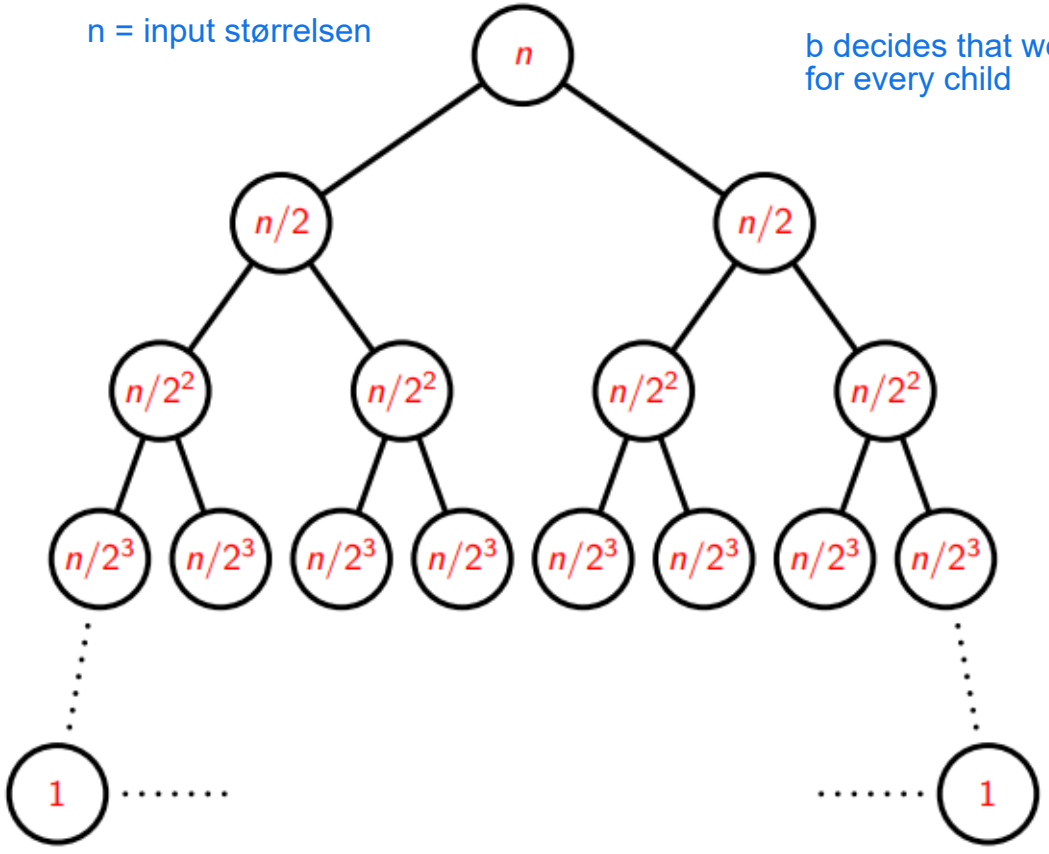
$$T(n) = 2T(n/2) + n$$

ved at bruge f(n), finder vi det lokale arbejde

$$a = 2,\ b = 2,\ f(x) = x$$

n = input størrelsen

b decides that we should divide by two, for every child

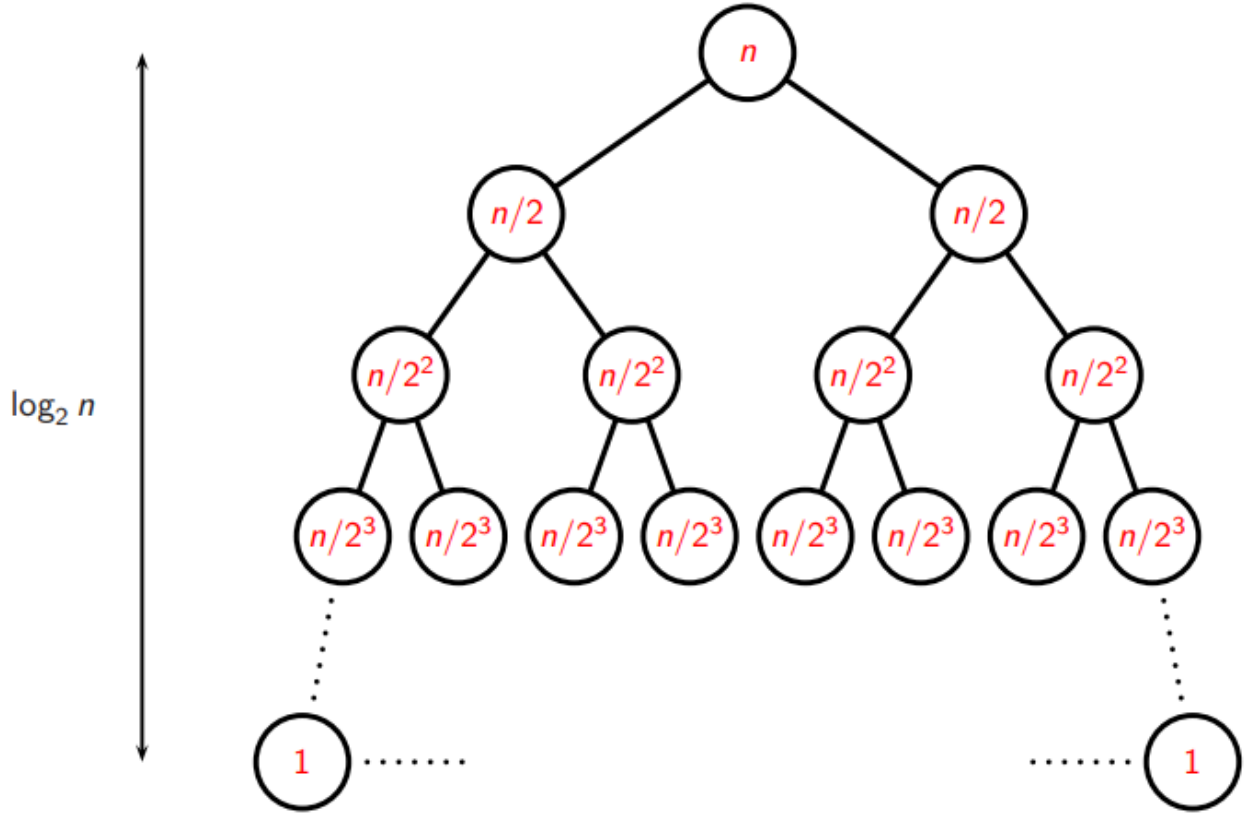


Insert input sizes into nodes (= n/b^{depth}).

Example1

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

$$a = 2,\ b = 2,\ f(x) = x$$

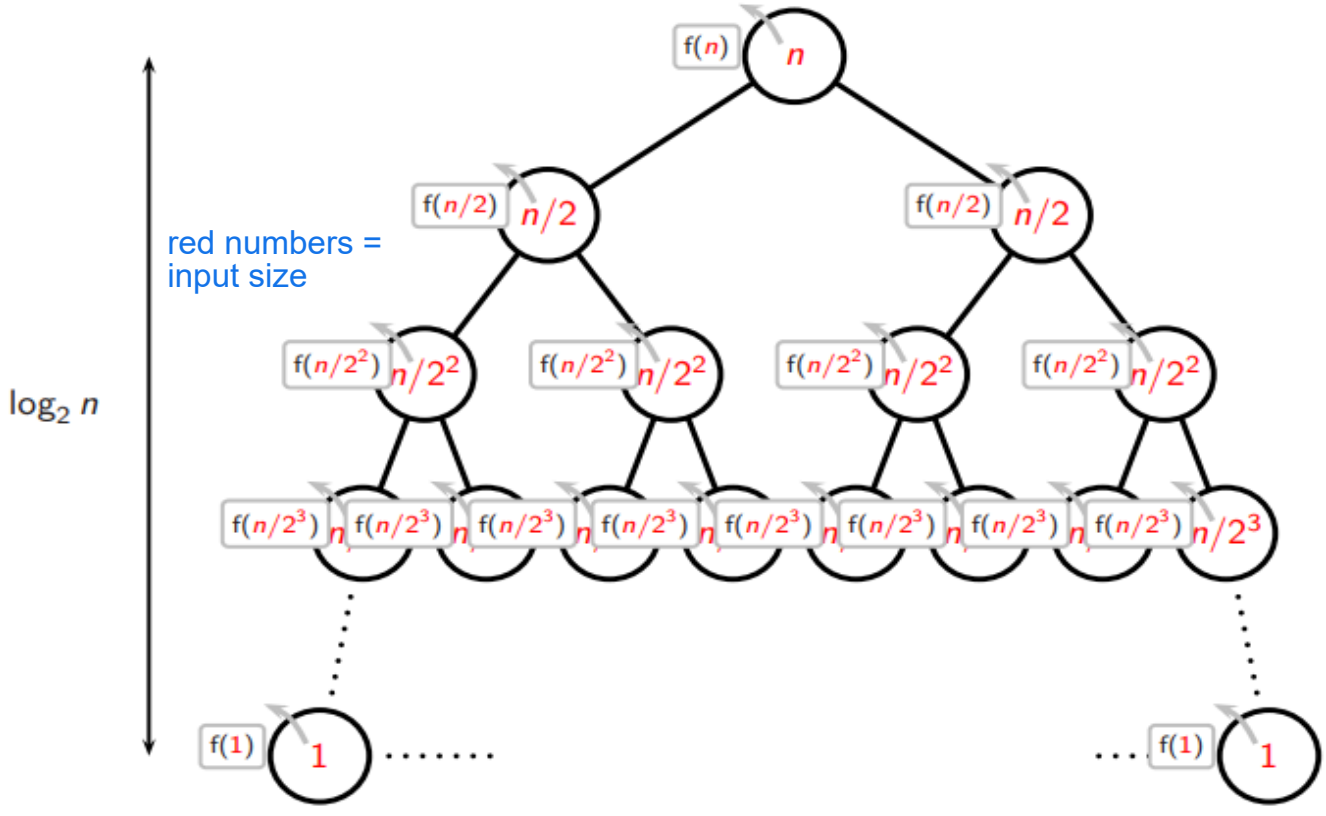


Find the height from $n/b^{\text{height}} = 1 \Leftrightarrow b^{\text{height}} = n \Leftrightarrow$ height = $\log_b n$.

Example1

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

$$a = 2,\ b = 2,\ f(x) = x$$



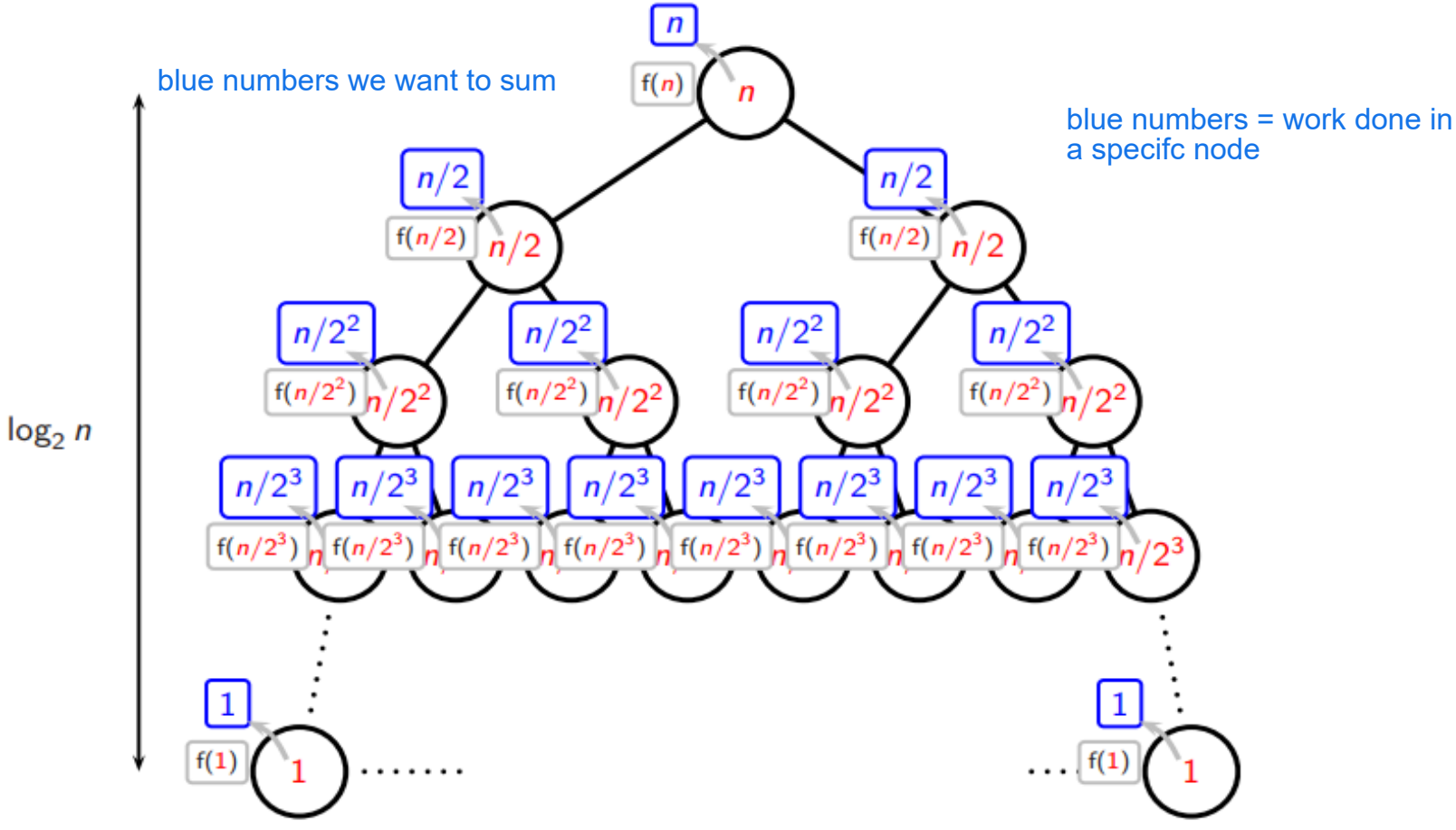
Use the f function to convert input size to work.

since f(x) = x, it is linear, and we simply get the same output

Example1

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

$$a = 2,\ b = 2,\ f(x) = x$$

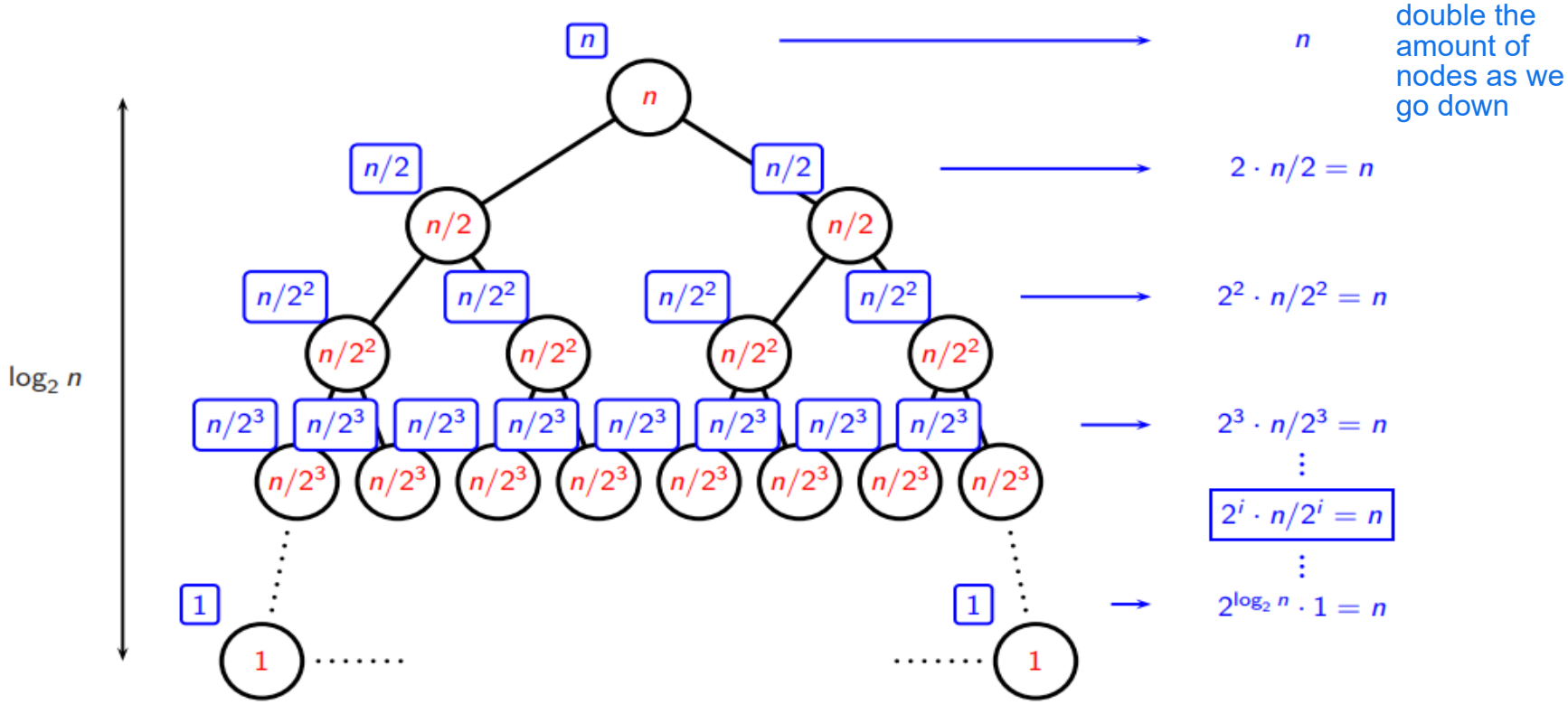


Use the function f to convert input size to work.

Example1

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

$$a = 2,\ b = 2,\ f(x) = x$$

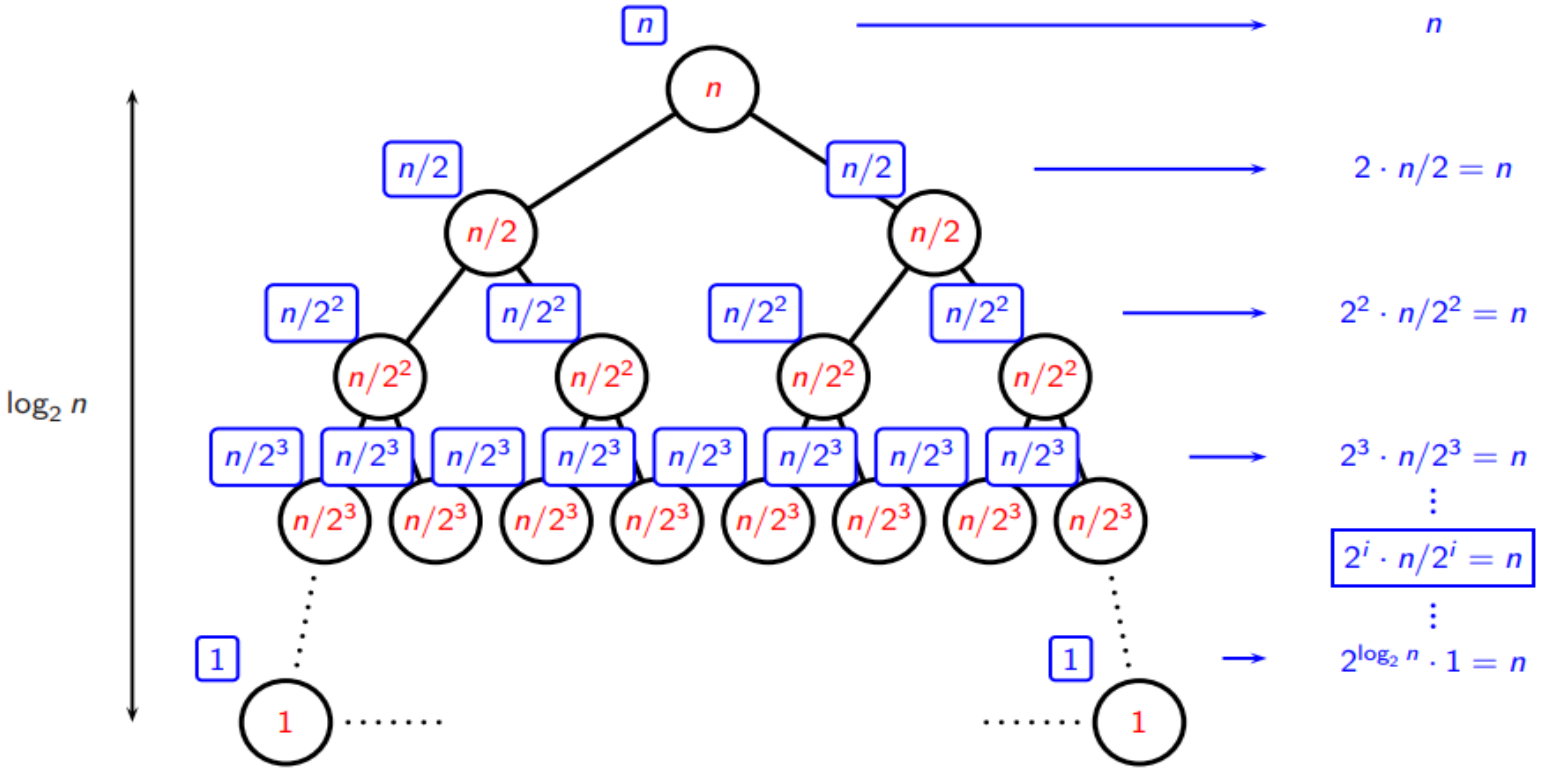


Sum each level together and express it as a function of the level number i.

Example 1

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

$$a = 2,\ b = 2,\ f(x) = x$$



Sum all layers together to find total work: $n \log n$.

Some mathematical facts for use in the next examples

As you know, the following theorem applies for c > 1:

$$1 + c + c^2 + .. + c^k = \frac{c^{k+1}-1}{c-1} = c^k . \frac{c-1/c^k}{c-1} = \Theta(c^k)$$

For c = 2 the theorem says that:

$$1 + 2 + 4 + 8 + .. + 2^k = 2^{k+1} - 1 = 2.2^k - 1 = \Theta(2^k)$$

This sentence should be remembered as follows:

> If the elements of a sum change exponentially, then the entire sum is dominated by the largest term.

Exponentially increasing/decreasing: largest term = last/first term.

Some mathematical facts for use in the next examples

Other facts:

- $(a^b)^c = a^{bc} = (a^c)^b$
- $a^{\log_b n} = n^{\log_b a}$
- $\log_b a = \log_c a / \log_c b$ (ex. $\log_b a = \ln a / \ln b$). this is used to find

The last fact gives us a way to calculate $\log_b a$ via a calculator (where the natural logarithm ln is found).

The last fact can also be written as: $\log_b x = (1 / \log_c b) \cdot \log_c x$ which shows that logarithms with different bases (here b and c) are a constant factor apart: $\log_b x = \Theta(\log_c x)$

Example 2

$$T(n) = 3T(n/2) + n$$

$a = 3$ 3 calls of the function it self

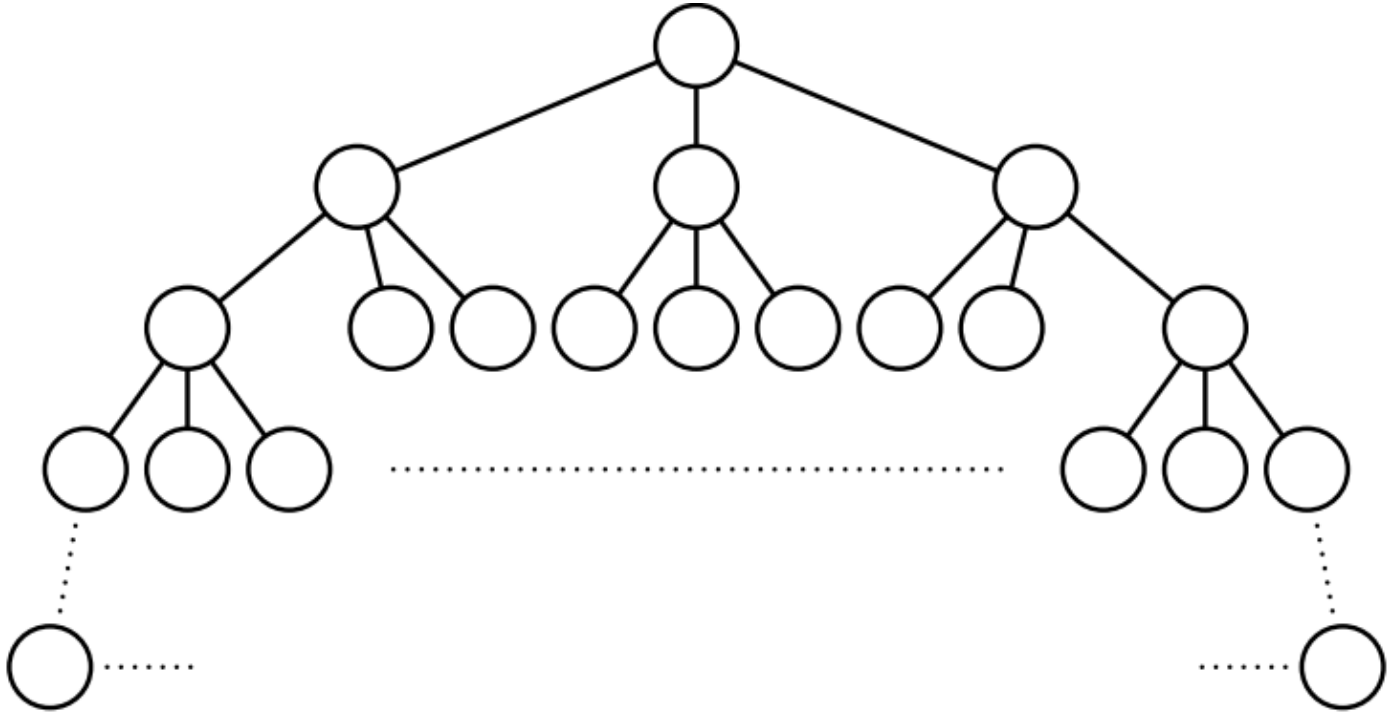
$b = 2$ Skriv tekst her

$f(n) = n$

Example 2

$T(n) = 3T(n/2) + n$

$a = 3,\ b = 2,\ f(x) = x$

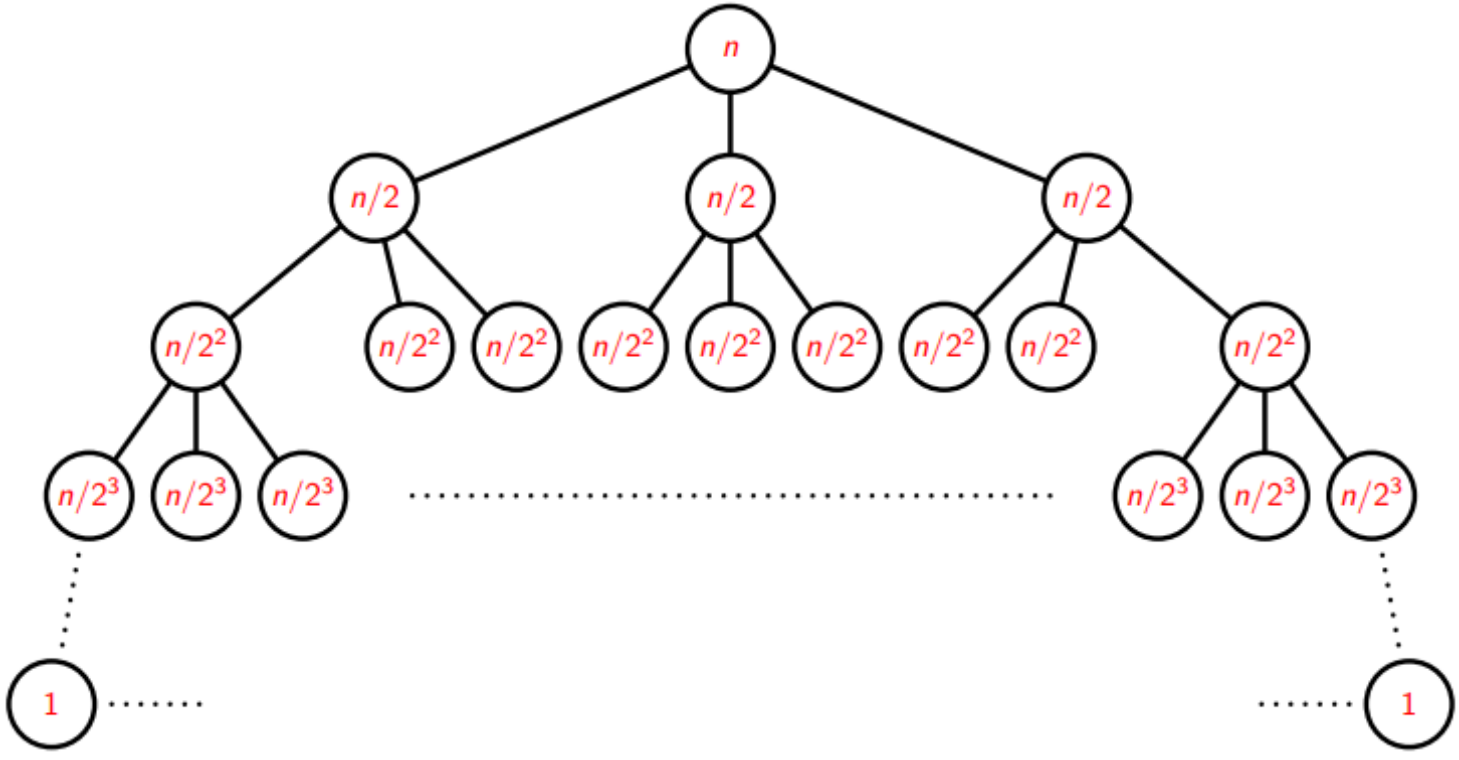


Draw a tree with $a = 3$.

Example 2

$$T(n) = 3T(n/2) + n$$

$$a = 3,\ b = 2,\ f(x) = x$$

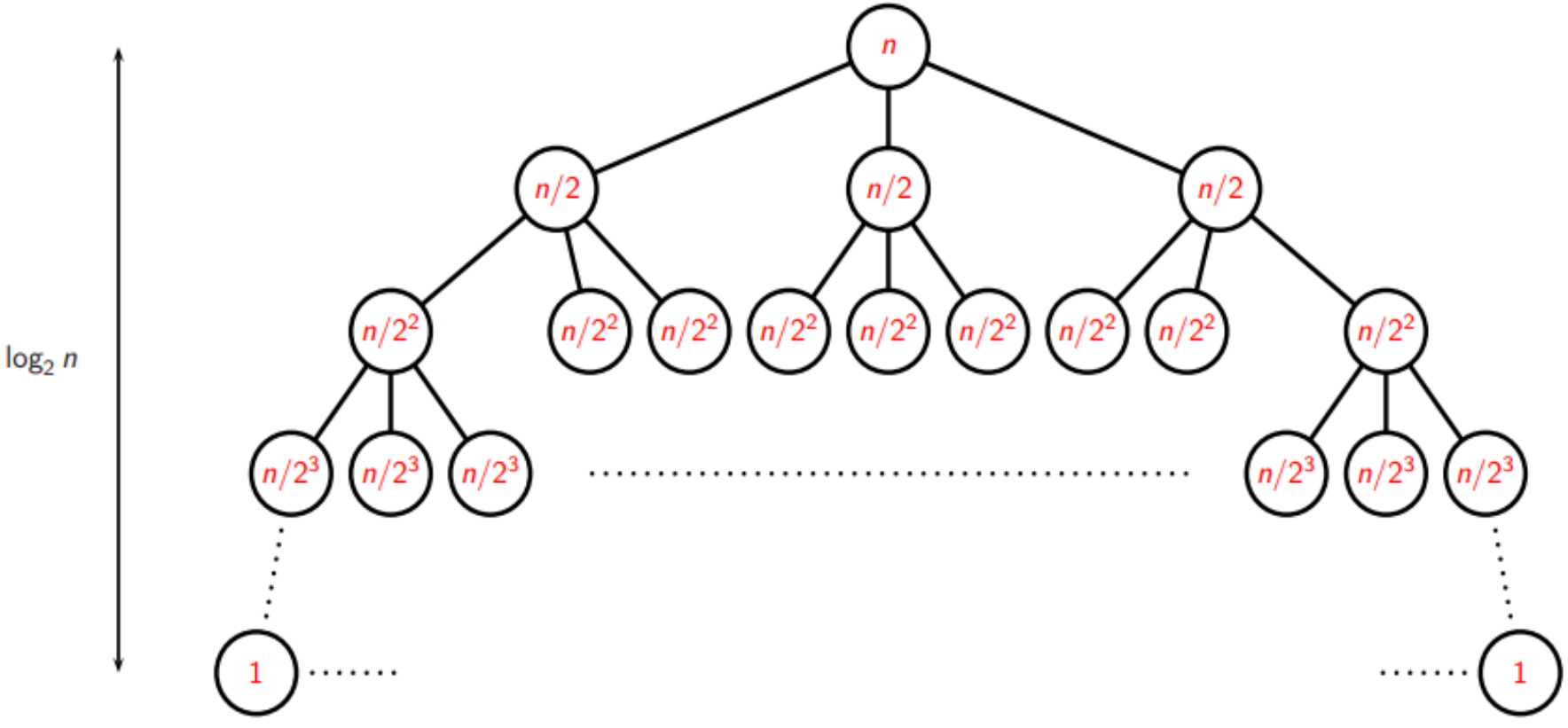


Insert input sizes into nodes (= n/b^{depth}).

Example 2

$$T(n) = 3T(n/2) + n$$

$$a = 3,\ b = 2,\ f(x) = x$$



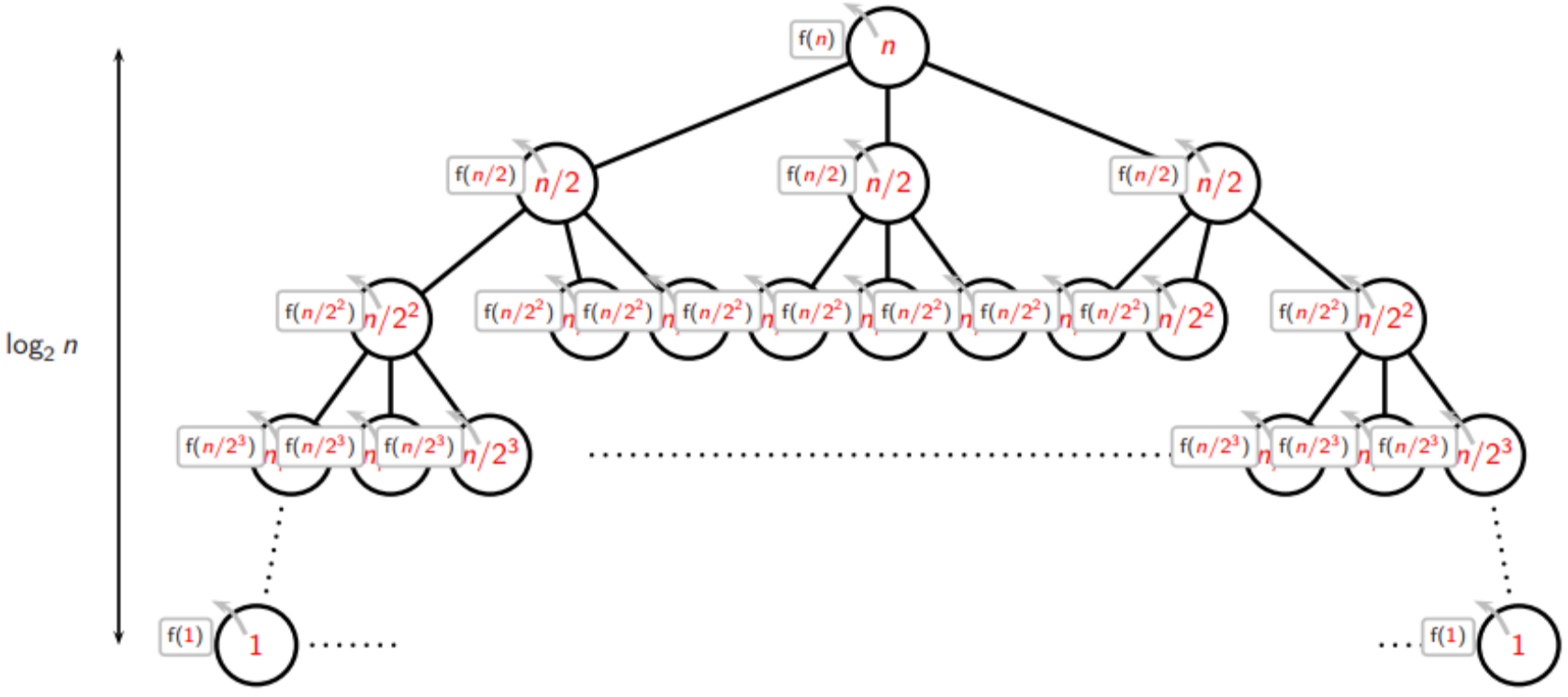
Find the height from $n/b^{\text{height}} = 1 \Leftrightarrow b^{\text{height}} = n \Leftrightarrow \text{height} = \log_b n$

for this: log_2 n

Example 2

$$T(n) = 3T(n/2) + n$$

$$a = 3,\ b = 2,\ f(x) = x$$

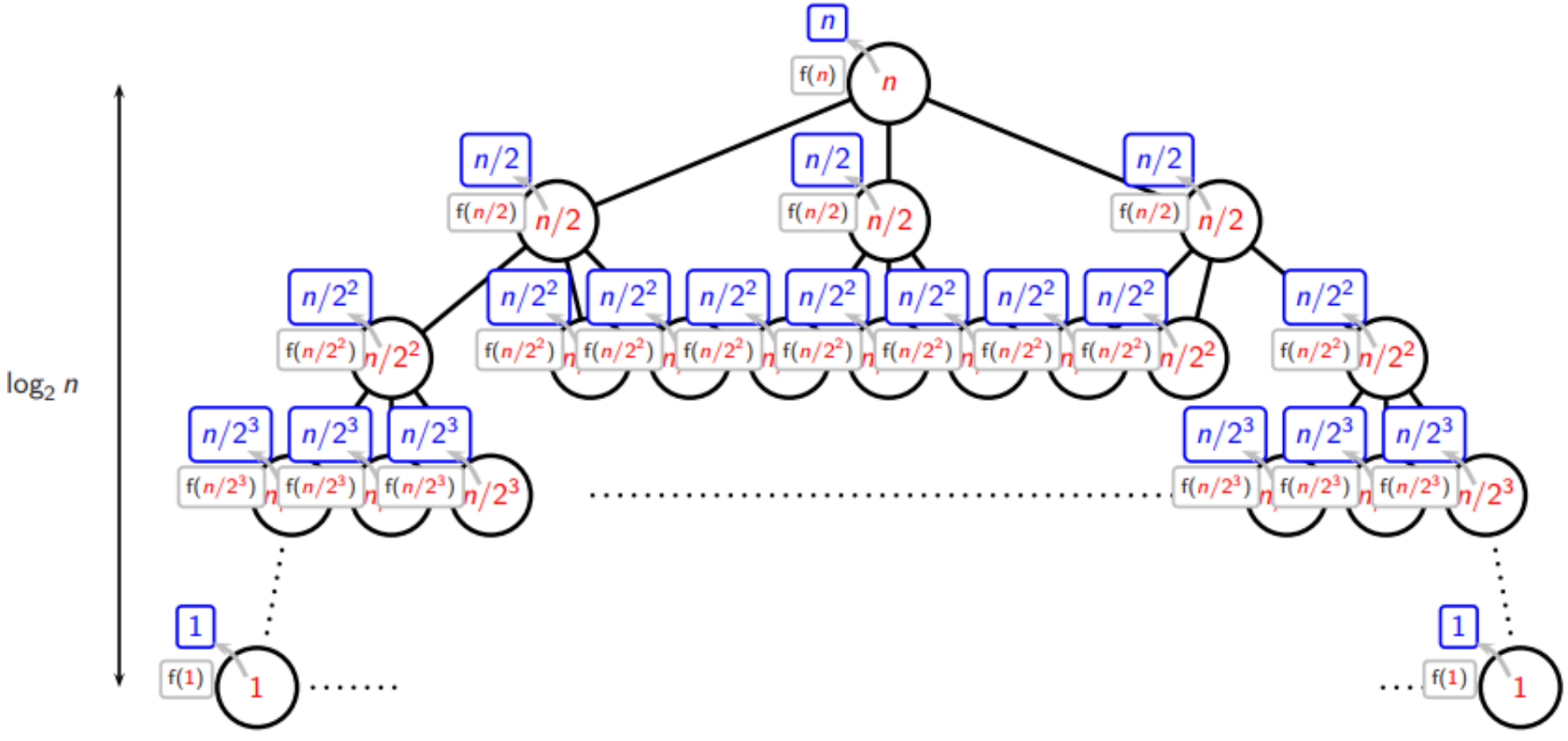


Use the function f to convert input size to work

Example 2

$$T(n) = 3T(n/2) + n$$

$$a = 3,\ b = 2,\ f(x) = x$$

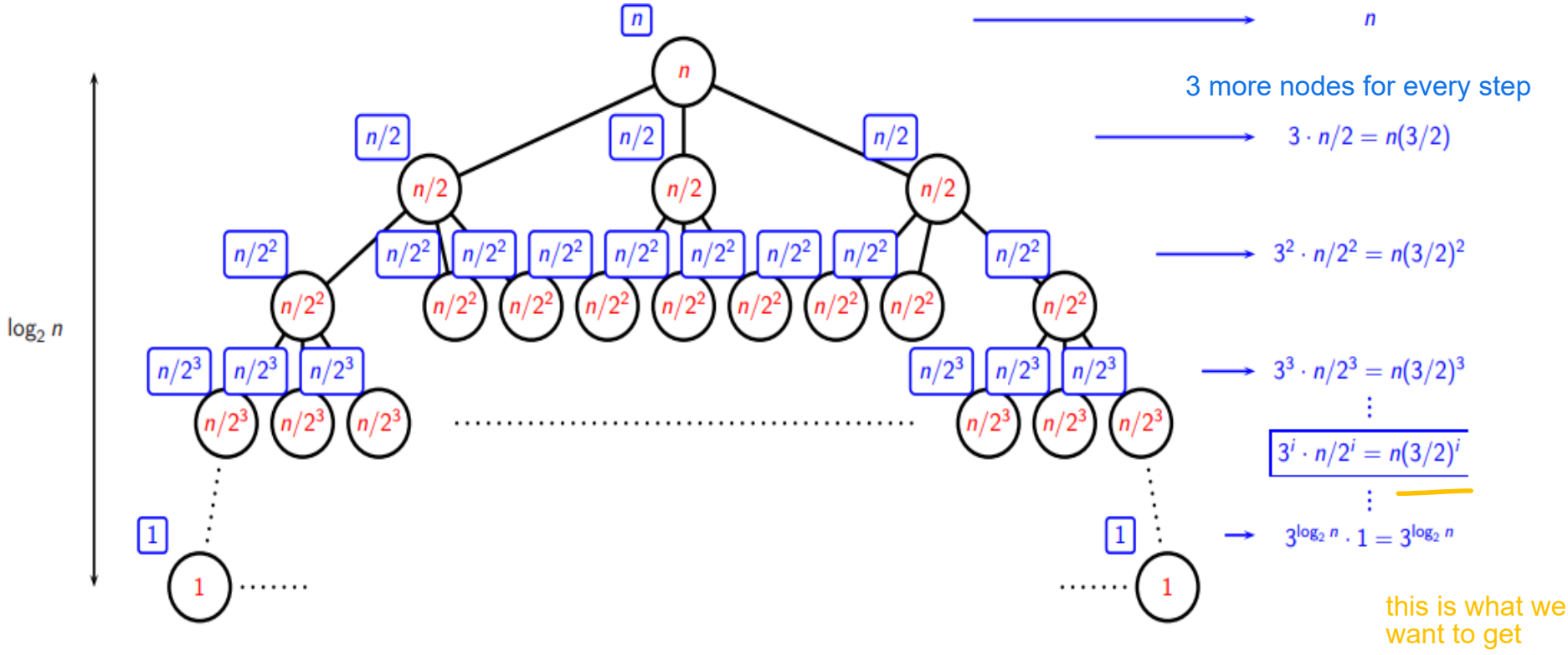


Use the function f to convert input size to work.

Example 2

$$T(n) = 3T(n/2) + n$$

$$a = 3,\ b = 2,\ f(x) = x$$

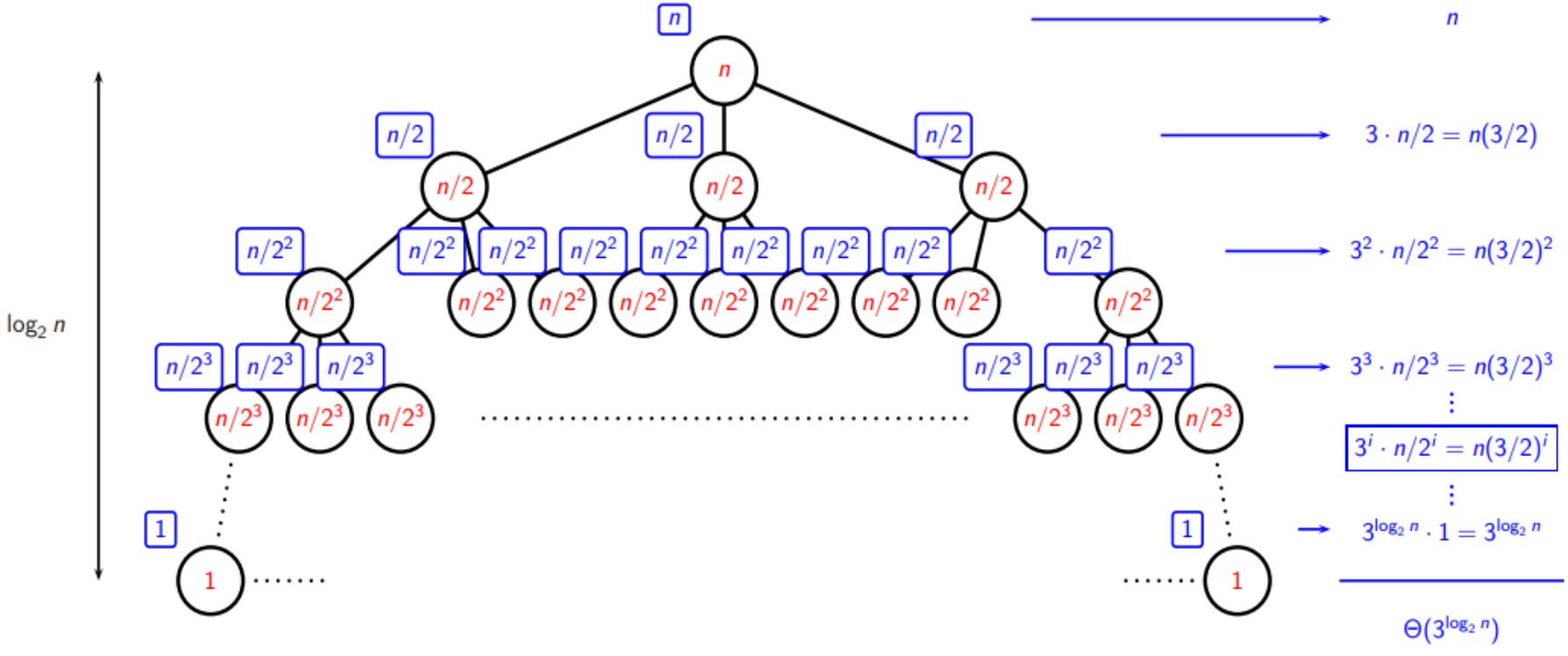


Sum each layer together and express as a function of the layer number *i*.

Example 2

$$T(n) = 3T(n/2) + n$$

$$a = 3,\ b = 2,\ f(x) = x$$



Exponentially increasing ⇒ last layer dominates ⇒ total work is $\Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3}) = \Theta(n^{\ln 3/\ln 2}) = \Theta(n^{1.5849\ldots})$.

Example 3

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

$$a = 3$$

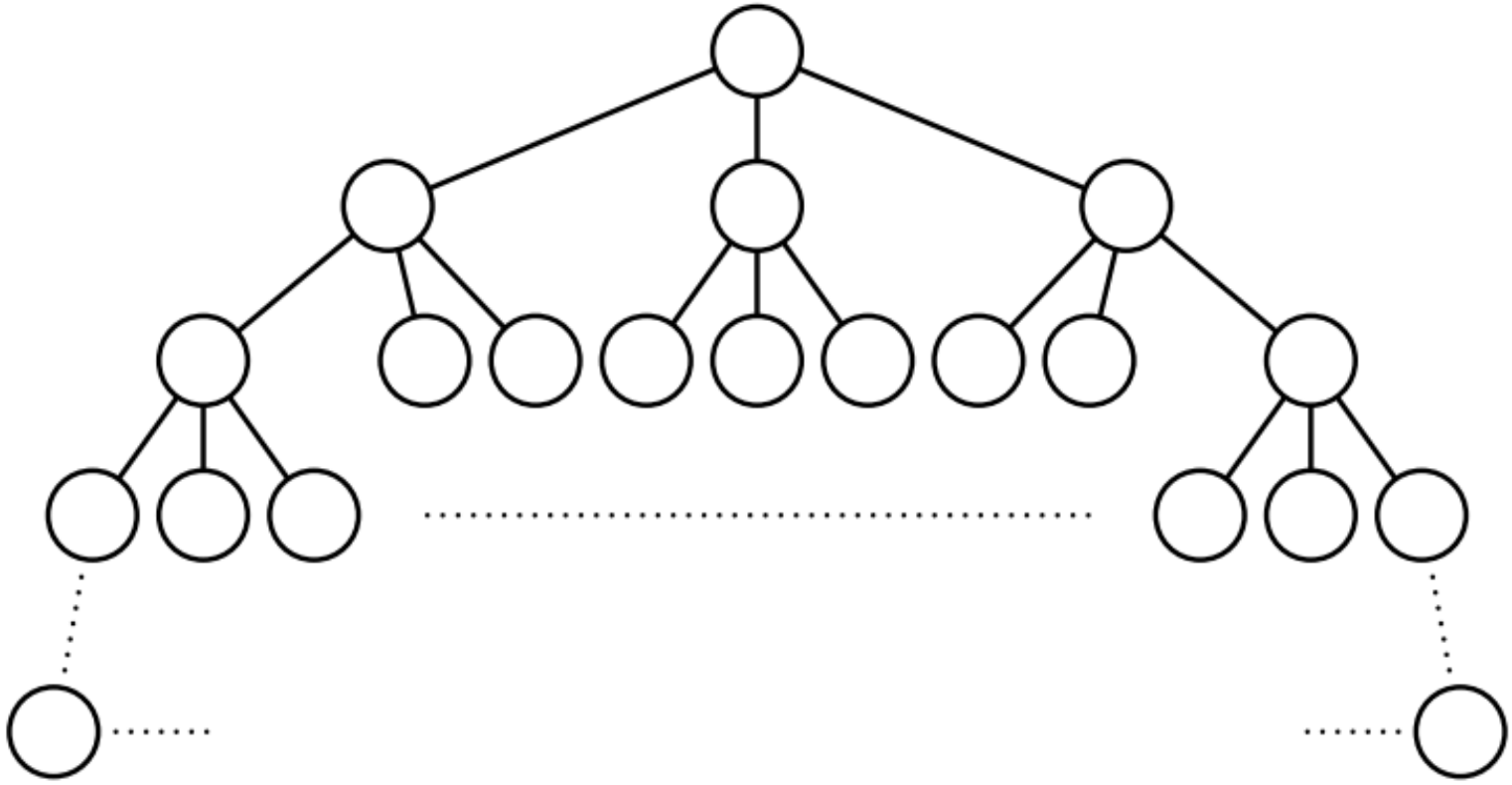
$$b = 4$$

$$f(n) = n^2$$

Example 3

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

$$a = 3,\ b = 4,\ f(x) = x^2$$

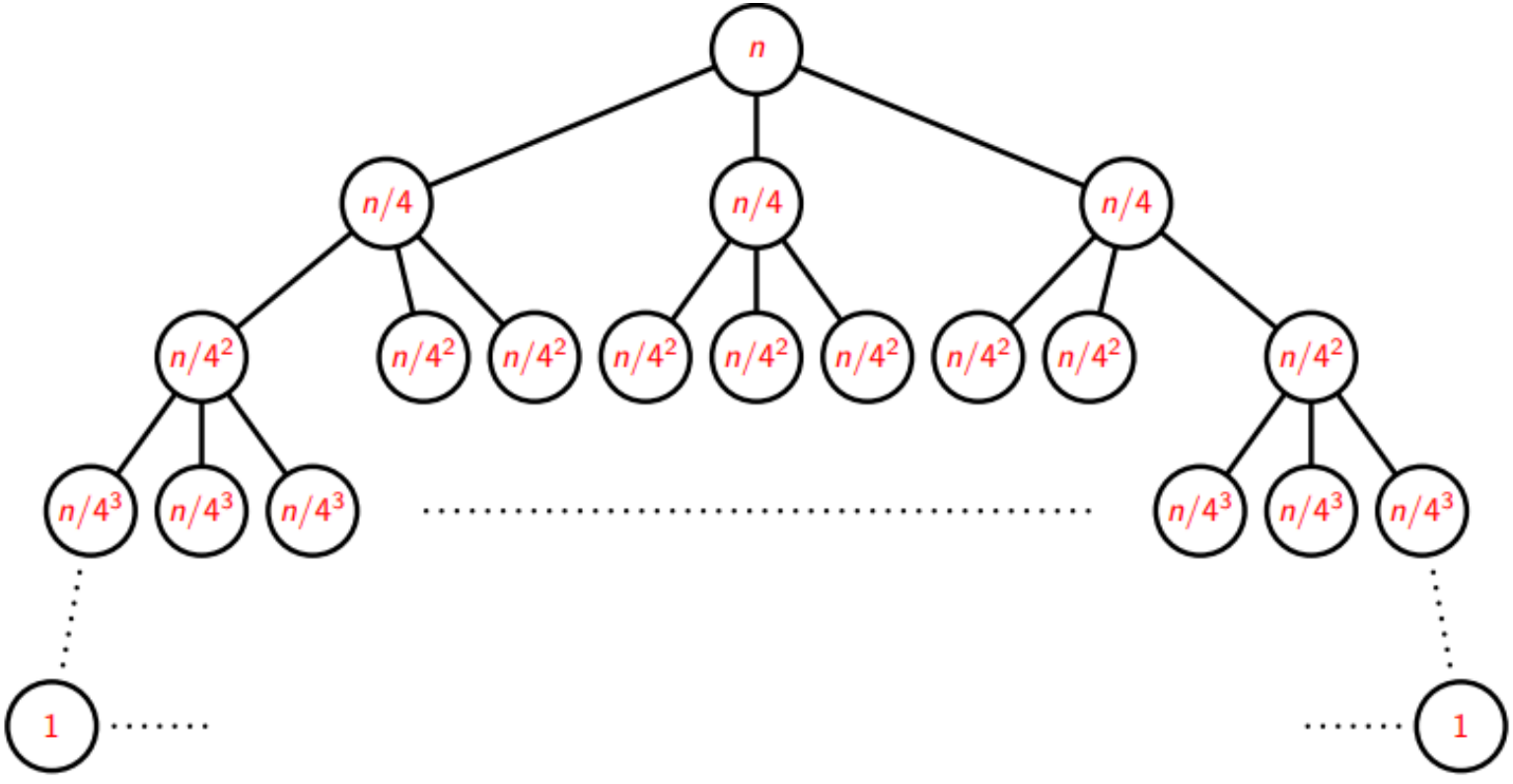


Draw a tree with $a = 3$.

Example 3

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

$$a = 3,\ b = 4,\ f(x) = x^2$$

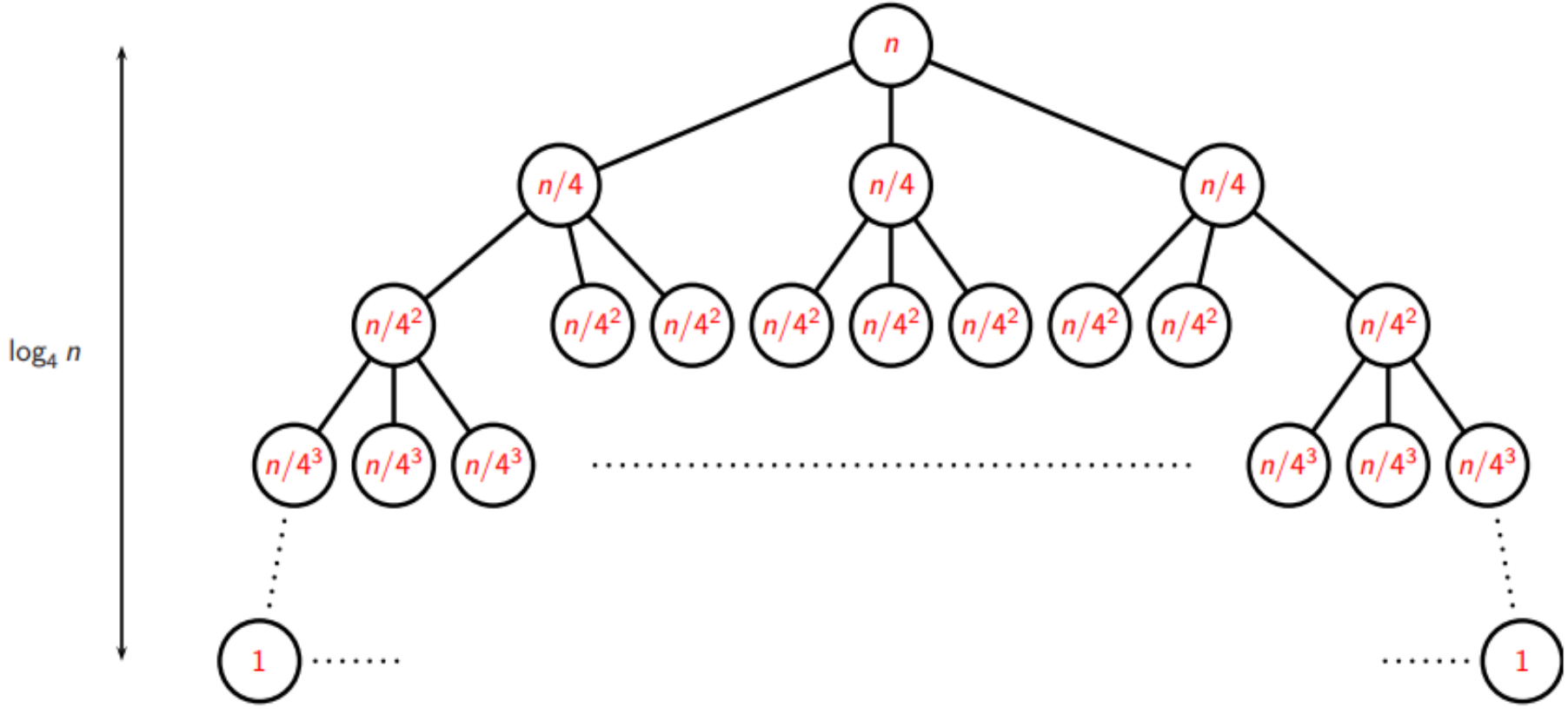


Insert input sizes into nodes (= n/b^{depth}).

Example 3

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

$$a = 3,\ b = 4,\ f(x) = x^2$$

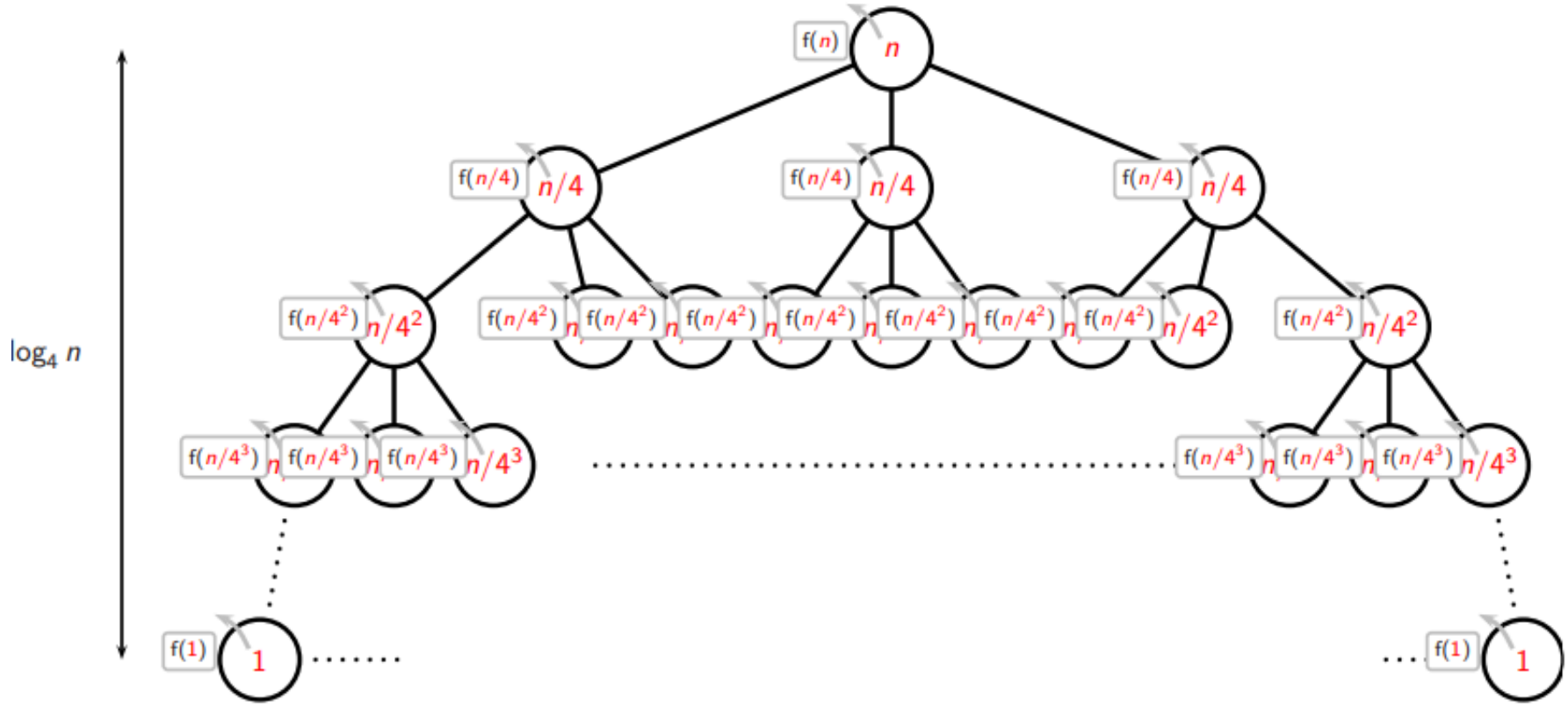


Find the height from $n/b^{\text{height}} = 1 \Leftrightarrow b^{\text{height}} = n \Leftrightarrow \text{height} = \log_b n$.

Example 3

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

$$a = 3,\ b = 4,\ f(x) = x^2$$

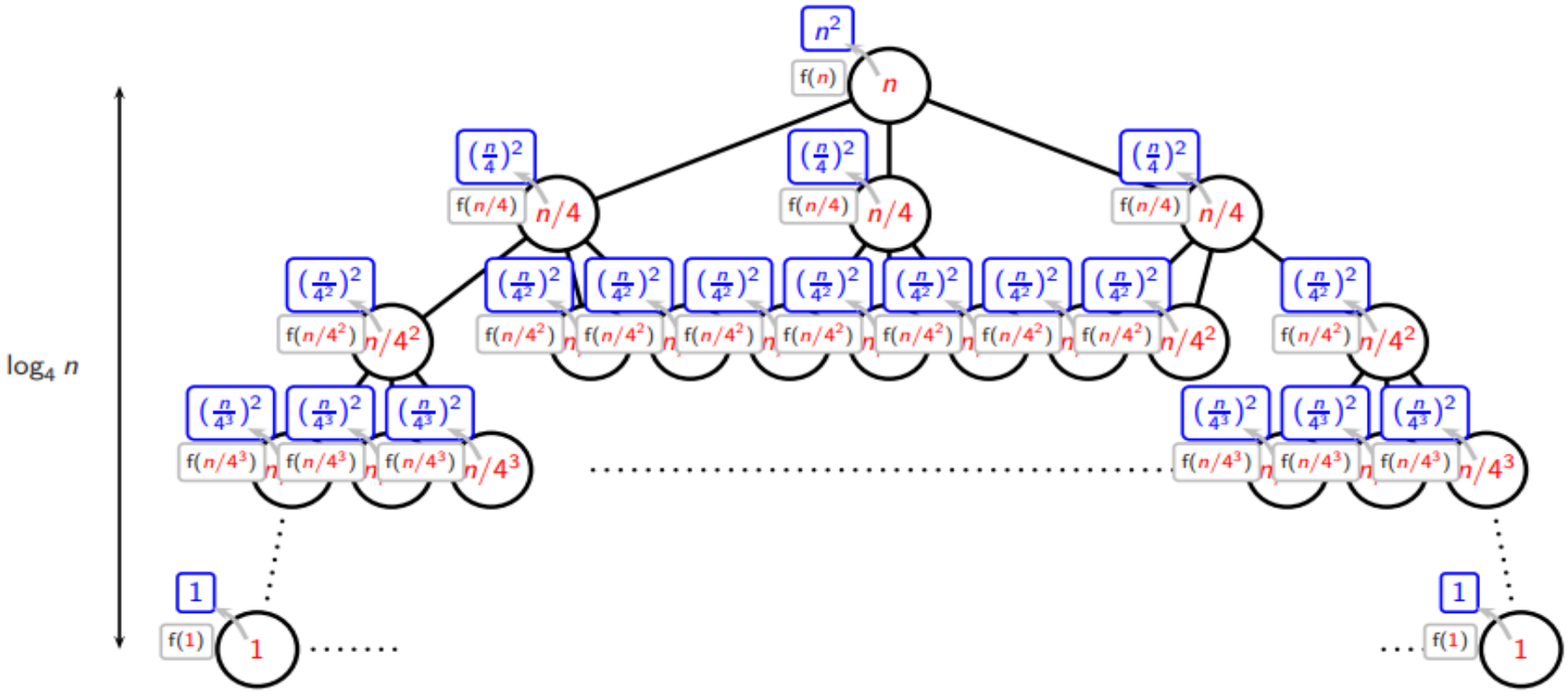


Use the f function to convert input size to work.

Example 3

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

$$a = 3,\ b = 4,\ f(x) = x^2$$

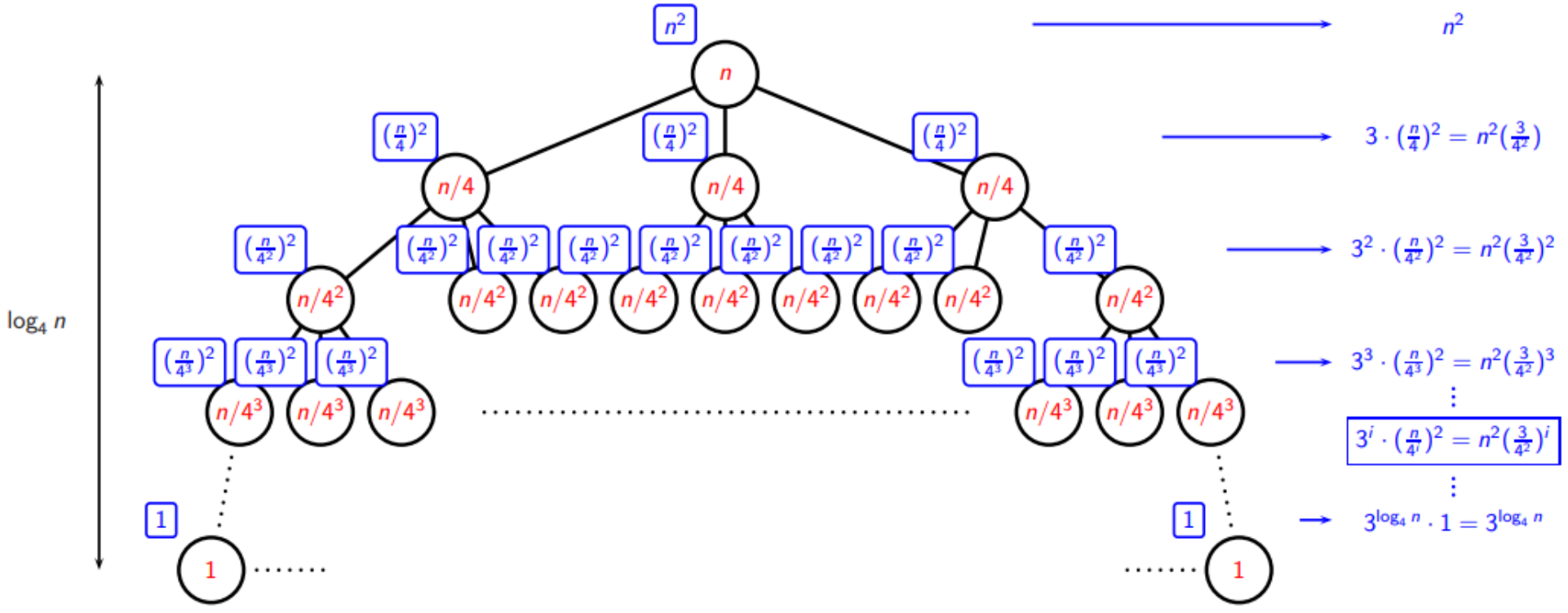


Use the function f to convert input size to work.

Example 3

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

$$a = 3,\ b = 4,\ f(x) = x^2$$

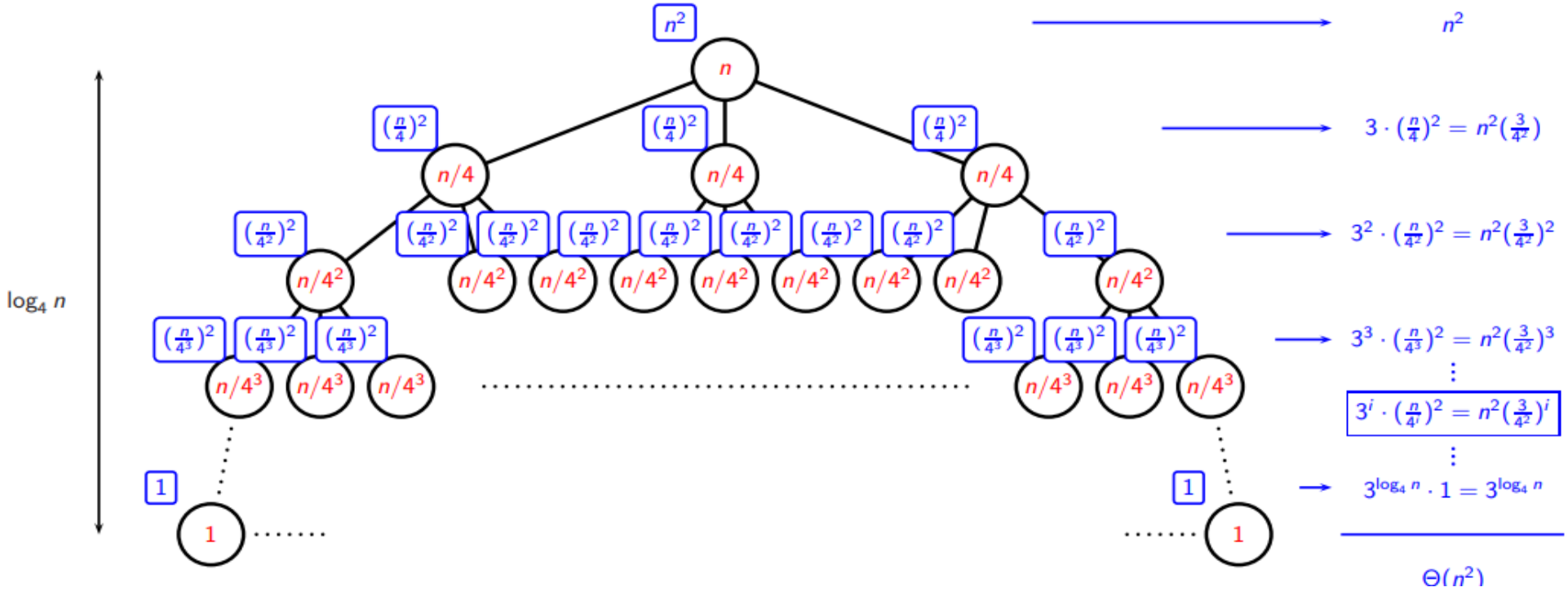


Sum each layer together and express as a function of the layer number *i*.

Example 3

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

$$a = 3,\ b = 4,\ f(x) = x^2$$



Exponentially decreasing ⇒ first layer dominates ⇒ total work is $\Theta(n^2)$.

Divide-and-Conquer examples

These three examples are representative. That is, one of the following often applies:

- All layers have approximately the same sum, whereby the total sum is the number of layers (the height of the tree) times this sum.
- The sum of the layers grows exponentially downwards through the layers, whereby the lowest layer dominates. To find the sum of this layer, one must know the height of the tree.
- The sum of the layers decreases exponentially downwards through the layers (= grows exponentially upwards through the layers), whereby the top layer dominates.

> A generic solution to these three cases is the content of the book's theorem on pages 102–3 [3rd edition: page 94], called the Master Theorem.

Most recursive algorithms are described by a recursion equation that fits into the Master Theorem. If it does not fit into the Master Theorem, one must try the recursion tree method directly (one can also do that if it fits into the Master Theorem, of course).

Master Theorem

MUST BE ON THIS FORM FOR US TO USE MASTER THEOREM

The recursion equation

this means we just subtract a very small amount

$$T(n) = aT(n/b) + f(n)$$

has the following solution, where $\alpha = \log_b a$:

1. If $f(n) = O(n^{\alpha-\epsilon})$ for $\epsilon > 0$ so: $\boldsymbol{T(n) = \Theta(n^{\alpha})}$. Upper bound
2. If $f(n) = \Theta(n^{\alpha}(\log n)^{k})$ for $k \geq 0$ so: $\boldsymbol{T(n) = \Theta(n^{\alpha}(\log n)^{k+1})}$. Tight fit
3. If $f(n) = \Omega(n^{\alpha+\epsilon})$ for $\epsilon > 0$ so: $\boldsymbol{T(n) = \Theta(f(n))}$.

add a small amount to the potens

Lower bound

Additional condition: In case 3 it must also be true that there exists $c < 1$ and n_0 such that $a \cdot f(n/b) \leq c \cdot f(n)$ when $n \geq n_0$.

> In short: the case is determined by the ratio between the growth rates of f(n) and $n^{\alpha}(\log n)^{k}$ (where $k \geq 0$).

If they are equal, we have case 2. If f(n) is smaller (by at least a factor n^{ϵ}), we have case 1. If f(n) is larger (by at least a factor n^{ϵ}), we have case 3.

(if the additional condition is met) usually always met

Master Theorem applied to the three examples

Example 1:

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

When doing master theorem you are asking does f(n), grow faster than n^log_b (a) ?

- a = 2
- $b = 2$
- $f(n) = n$
- $\alpha = \log_2 2 = 1$
- $f(n) = n = \Theta(n^1) = \Theta(n^\alpha (\log n)^0)$ [that: $k = 0$]

f(n) grows the same as n^a

theorem says this is the result

So we are in case 2, so the following applies:

$$T(n) = \Theta(n^\alpha (\log n)^{0+1}) = \Theta(n \log n).$$

Master Theorem applied to the three examples

Example 2:

$$T(n) = 3T(n/2) + n$$

- $a = 3$
- $b = 2$
- $f(n) = n$
- $\alpha = \log_2 3 = \ln(3)/\ln(2) = 1.5849\ldots$
- $f(n) = n = O(n^{1.5849\cdots-\epsilon}) = O(n^{\alpha-\epsilon})$ for example $\epsilon = 0.1$.

to get n^1, we need to subtract something from n^1.5849

So we are in case 1, so the following applies:

$$T(n) = \Theta(n^{\alpha}) = \Theta(n^{1.5849\ldots}).$$

Master Theorem applied to the three examples

Example 3:

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

- $a = 3$
- $b = 4$
- $f(n) = n^2$
- $\alpha = \log_4 3 = \ln(3)/\ln(4) = 0.7924\ldots$
- $f(n) = n^2 = \Omega(n^{0.7924\cdots+\epsilon}) = \Omega(n^{\alpha+\epsilon})$ for example $\epsilon = 0.1$.

Here we need to add something to n^a to get n ^2

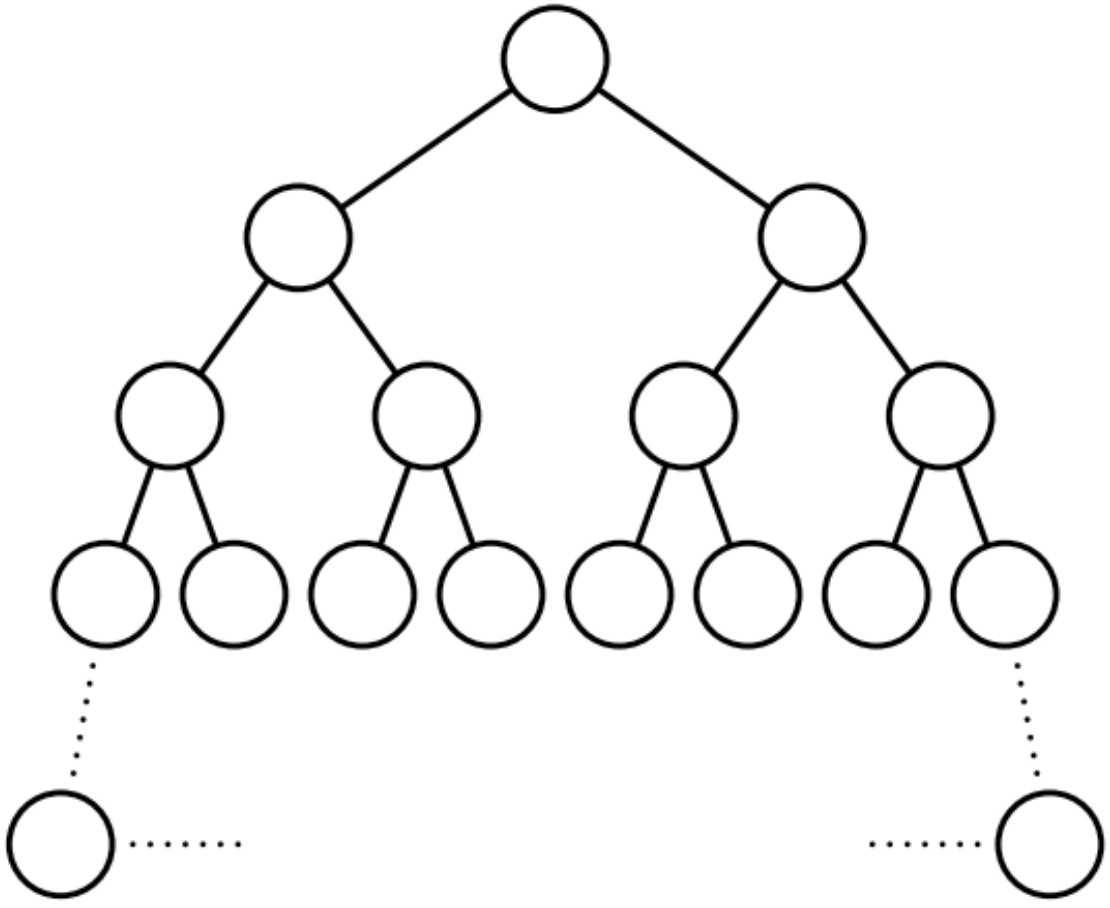
So we are in case 3, so the following applies: $T(n) = \Theta(f(n)) = \Theta(n^2)$.

In case 3, however, we must also check the additional condition: With $c = 3/16 < 1$ and $n_0 = 1$, it is satisfied that $a \cdot f(n/b) = 3(n/4)^2 = 3/16 \cdot n^2 \leq c \cdot f(n) = c \cdot n^2$ for all $n \geq n_0$.

can we find a constant less than 1?, yes 3/16. And find a n_0, yes typically 1.
so we get 3/16*1^2 <= 3/16*1^2.

Example 4

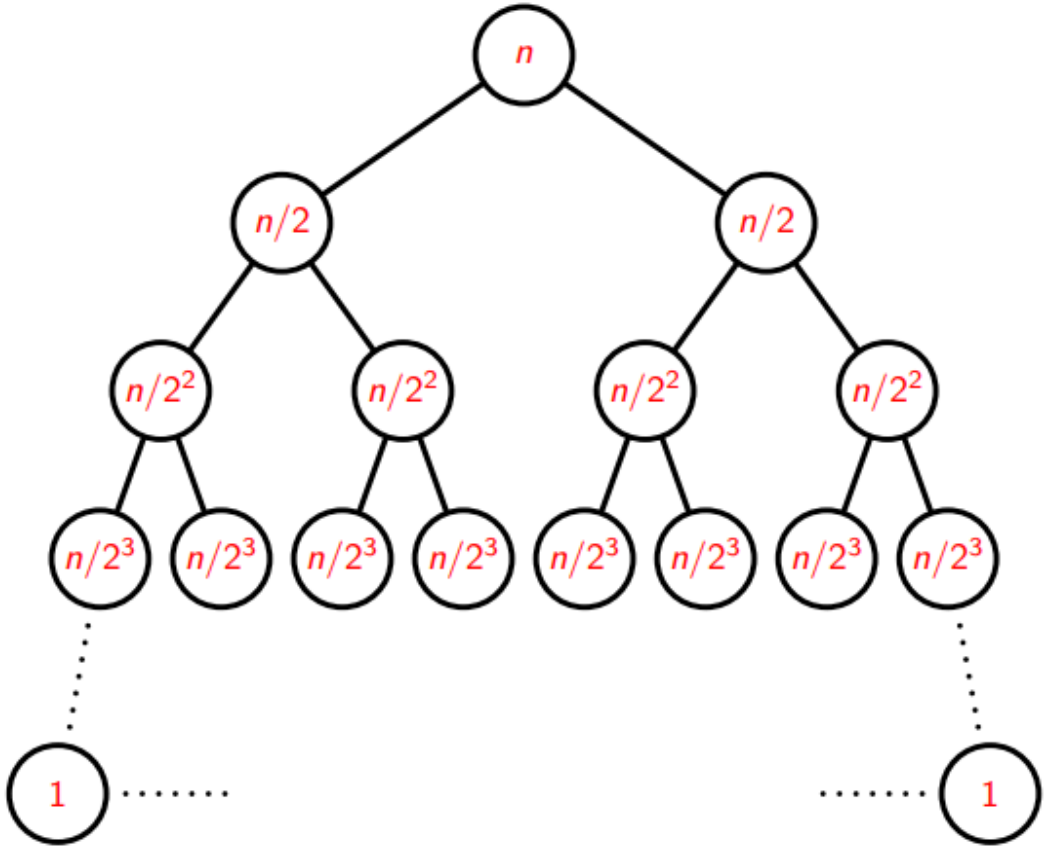
$T(n) = 2T(n/2) + n \log n$, $a = 2$, $b = 2$, $f(x) = x \log x$



Draw a tree with $a = 2$.

Example 4

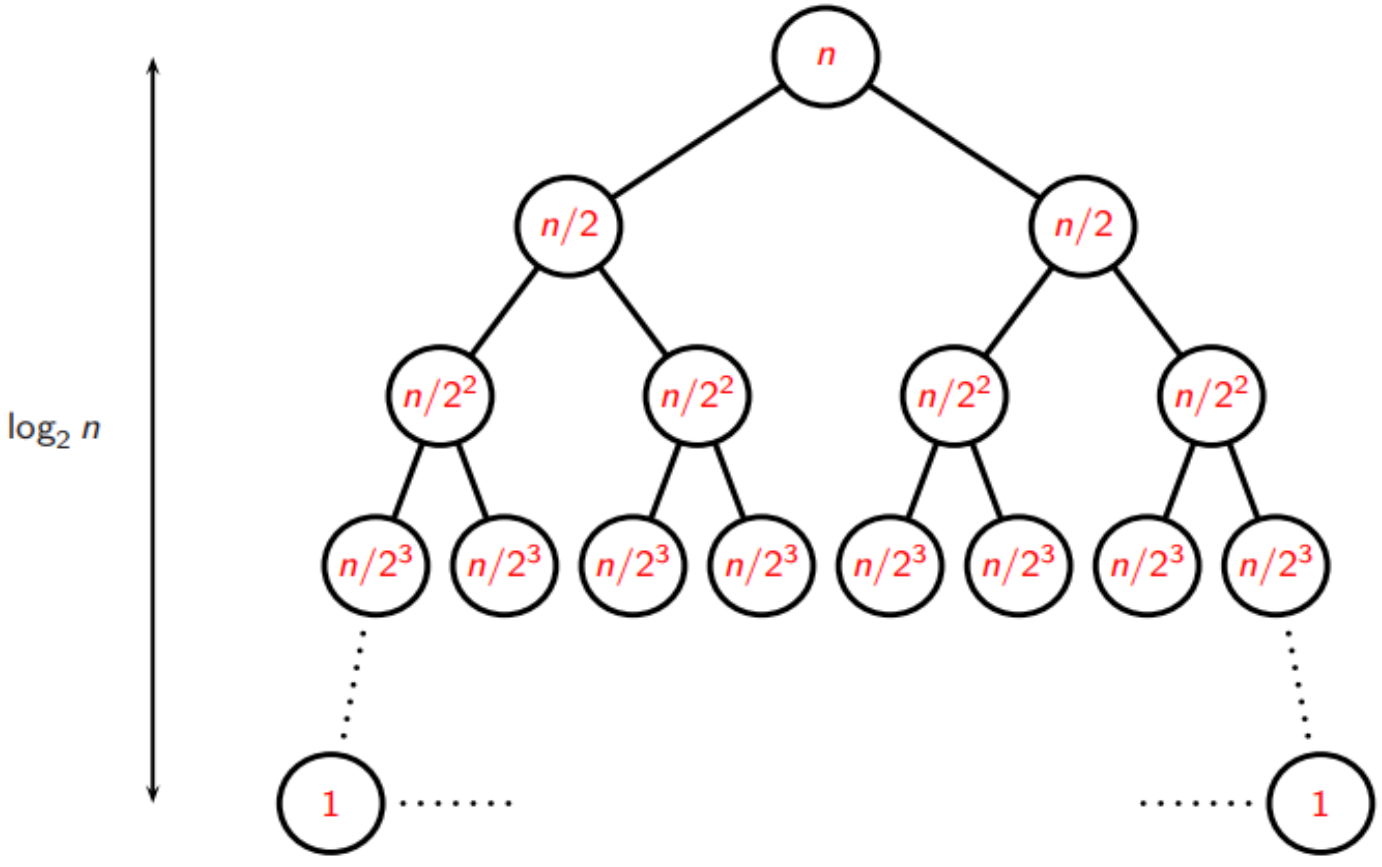
$T(n) = 2T(n/2) + n \log n$, $a = 2$, $b = 2$, $f(x) = x \log x$



Insert input sizes into nodes (= n/b^{depth}).

Example 4

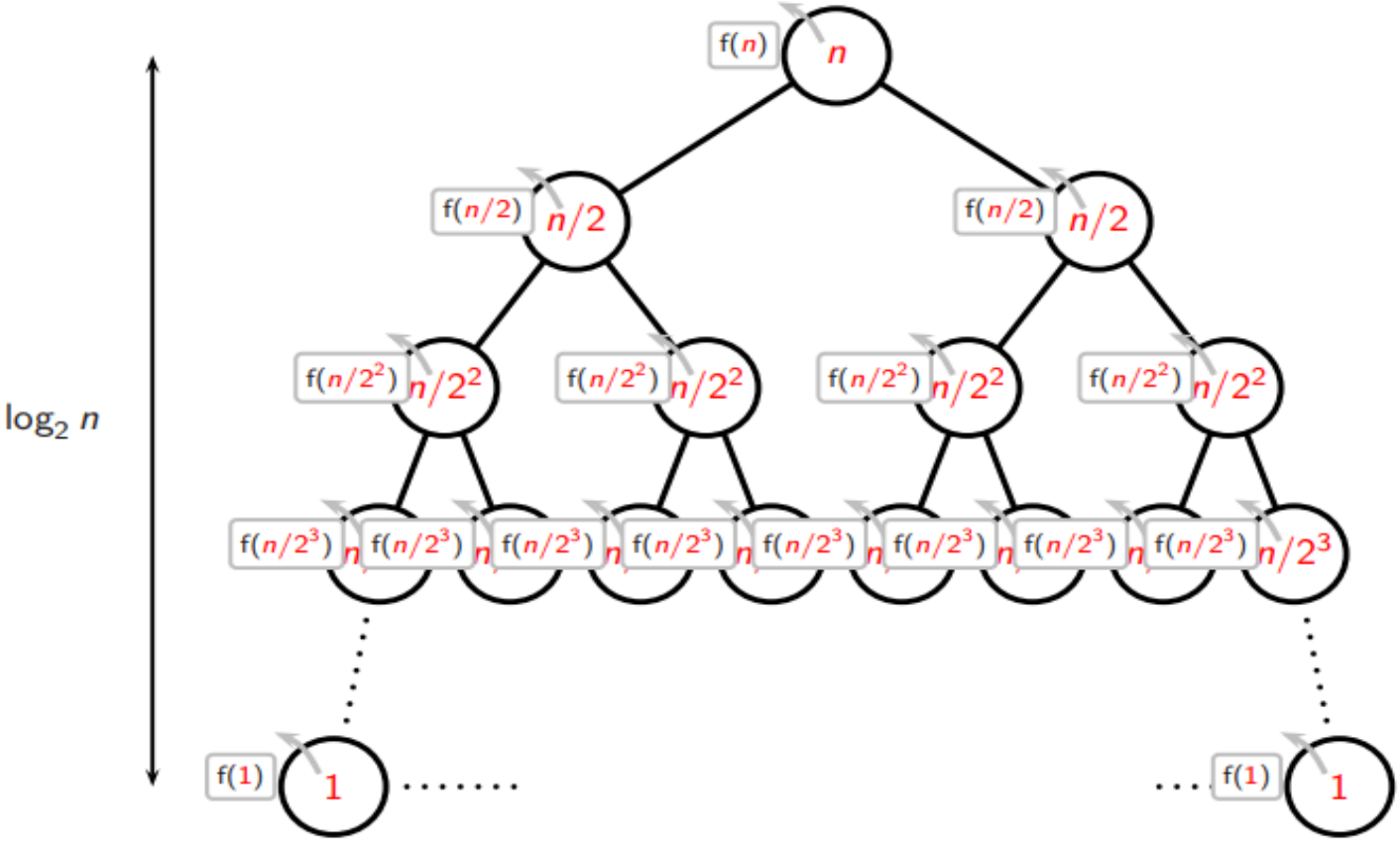
$T(n) = 2T(n/2) + n \log n$, $a = 2$, $b = 2$, $f(x) = x \log x$



Find the height from $n/b^{\text{height}} = 1 \Leftrightarrow b^{\text{height}} = n \Leftrightarrow \textit{height} = \log_b n$.

Example 4

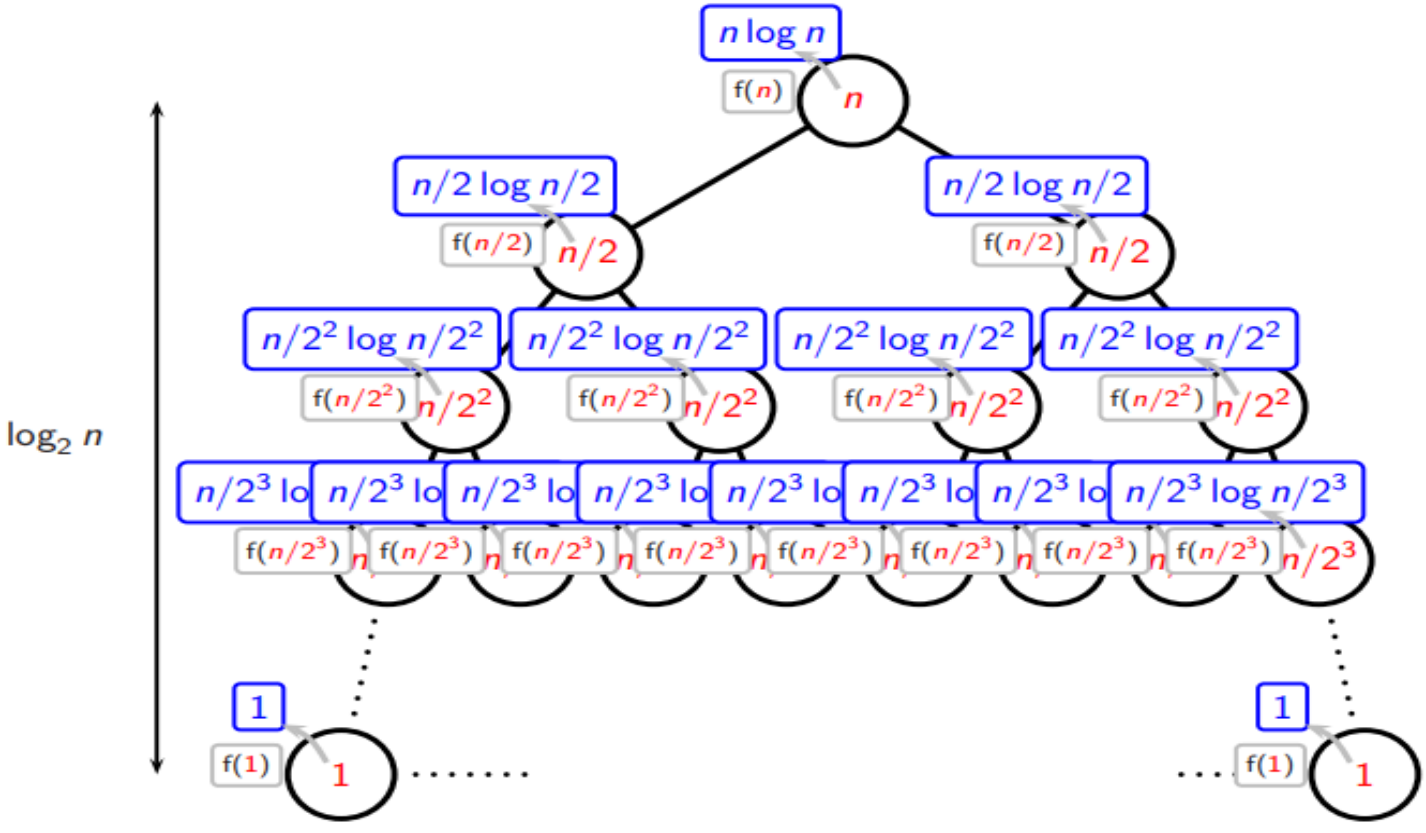
$T(n) = 2T(n/2) + n \log n$, $a = 2$, $b = 2$, $f(x) = x \log x$



Use the function f to convert input size to work.

Example 4

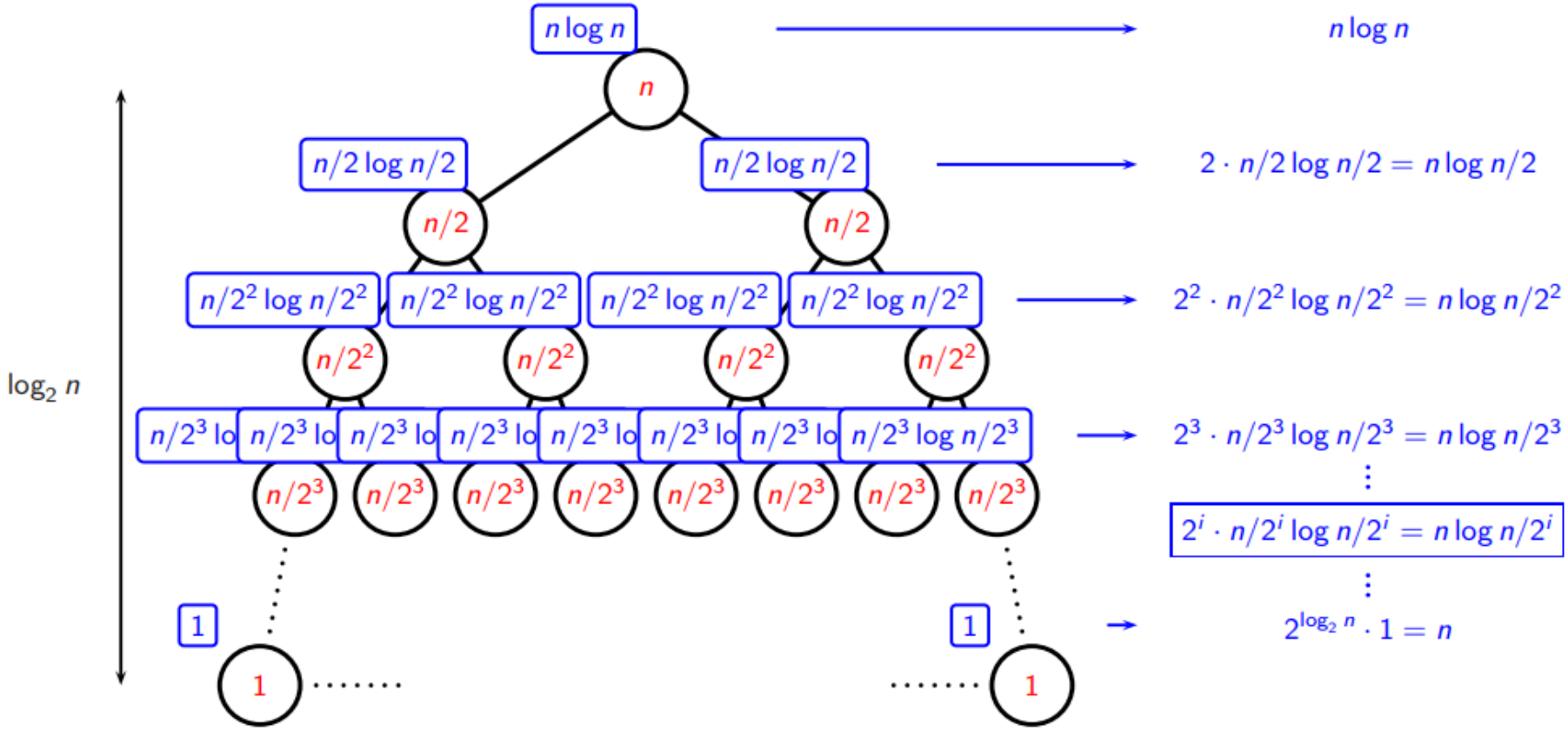
$T(n) = 2T(n/2) + n \log n$, $a = 2$, $b = 2$, $f(x) = x \log x$



Use the f function to convert input size to work.

Example 4

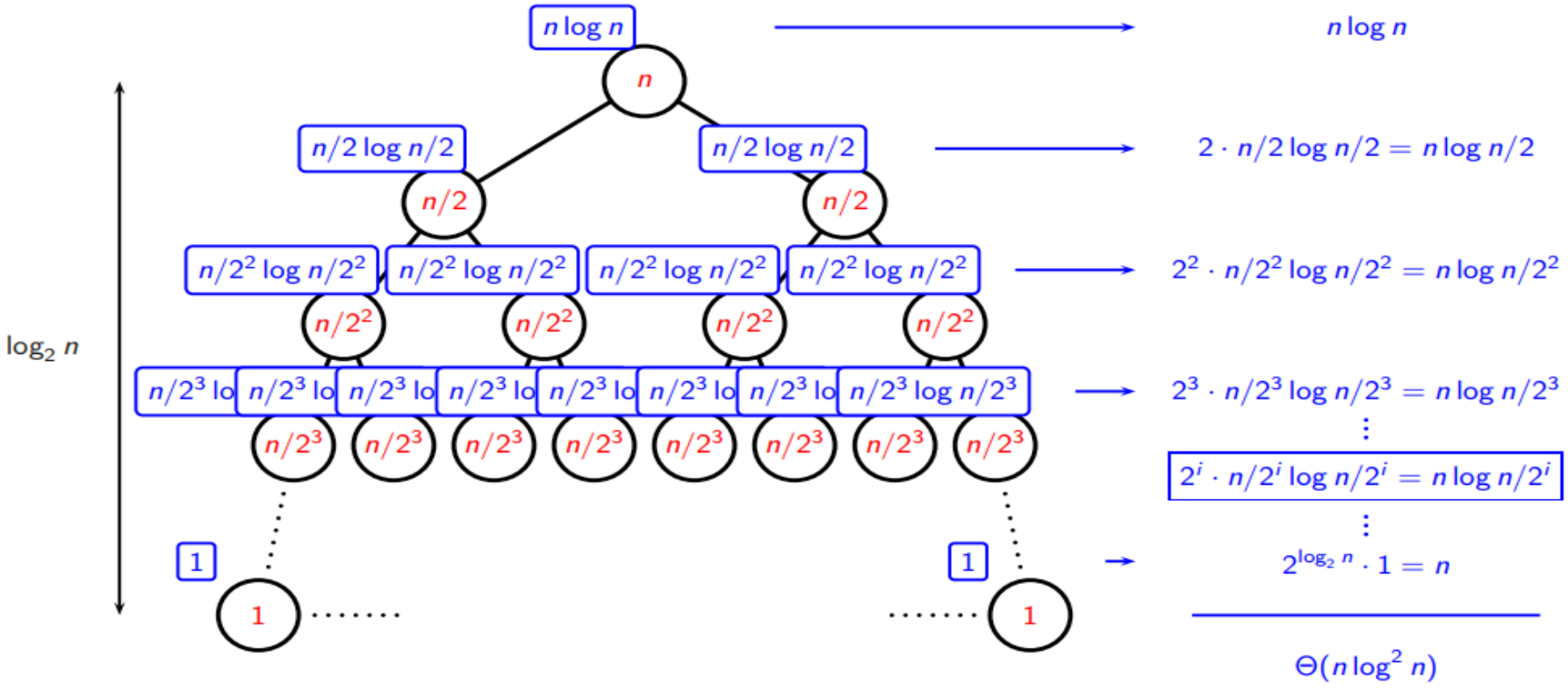
$T(n) = 2T(n/2) + n \log n$, $a = 2$, $b = 2$, $f(x) = x \log x$



Sum each layer and express it as a function of layer number i.

Example 4

$T(n) = 2T(n/2) + n \log n$, $a = 2, b = 2, f(x) = x \log x$



Sum all layers together to find total work. The answer is $\Theta(n \log^2 n)$, which we will argue on the next page.

Example 4

We have seen that the ith layer in the recursion tree does n log($n/2^i$) work, which can be written as n(log(n) − log(2^i)) = n(log(n) − i). This expression decreases as i increases, so that all layers do at most the same work as the first layer (i = 0). The first layer does n log n work and there are log n layers in total, so the total work is $O(n \log^2 n)$.

We now look at the top half of the layers, i.e. at layer i for
i = 1, 2, . . . , k =(½)log n.

The work for layer i decreases as i increases, so that each of these layers does at least the same work as the last one (layer k). The work for layer k is

$$n\,(\log(n) - k) = n\,(\log(n) - \tfrac{1}{2}\log n) = \tfrac{1}{2}\, n \log n.$$

There are therefore k =½ log n layers, each doing at least ½ n log n work. The total work is therefore $\Omega(n \log^2 n)$.

In total, we have shown that the total work is $\Theta(n \log^2 n)$.

Master Theorem used on example 4

Example 4:

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

- $a = 2$
- $b = 2$
- $f(n) = n \log n$
- $\alpha = \log_2 2 = 1$
- $f(n) = n \log n = \Theta(n^1(\log n)^1) = \Theta(n^\alpha(\log n)^1)$ [that is. $k = 1$]

So we are in case 2, so the following applies: $T(n) = \Theta(n^\alpha(\log n)^{1+1}) = \Theta(n(\log n)^2)$. [Note: $(\log n)^2$ is most often written as $\log^2 n$]

Master Theorem cannot always be used

The Master Theorem can be seen as a pre-computed recursion tree method that covers many recursion equations.

Regarding the solution of recursion equations, it is sufficient to be able to do the following for the exam:

- Use the Master Theorem in situations where it can be used.
- Recognize situations where the Master Theorem cannot be used.

For the second point, we now give an example of a recursion equation where the Master Theorem cannot be used.

For the case shown, it is actually possible to implement the recursion tree method and find the running time that way. We also show how (but this is not the syllabus for the exam).

Example 5

two recursive calls

$$T(n) = T(n/3) + T(2n/3) + n$$

This recursion equation gives the running time of a recursive algorithm that does O(n) local work and makes two recursive calls, one of size n/3 and one of size 2n/3.

Unfortunately, this recursion equation is not of the type that the Master Theorem deals with. For them, all recursive calls must have the same size (n/b):

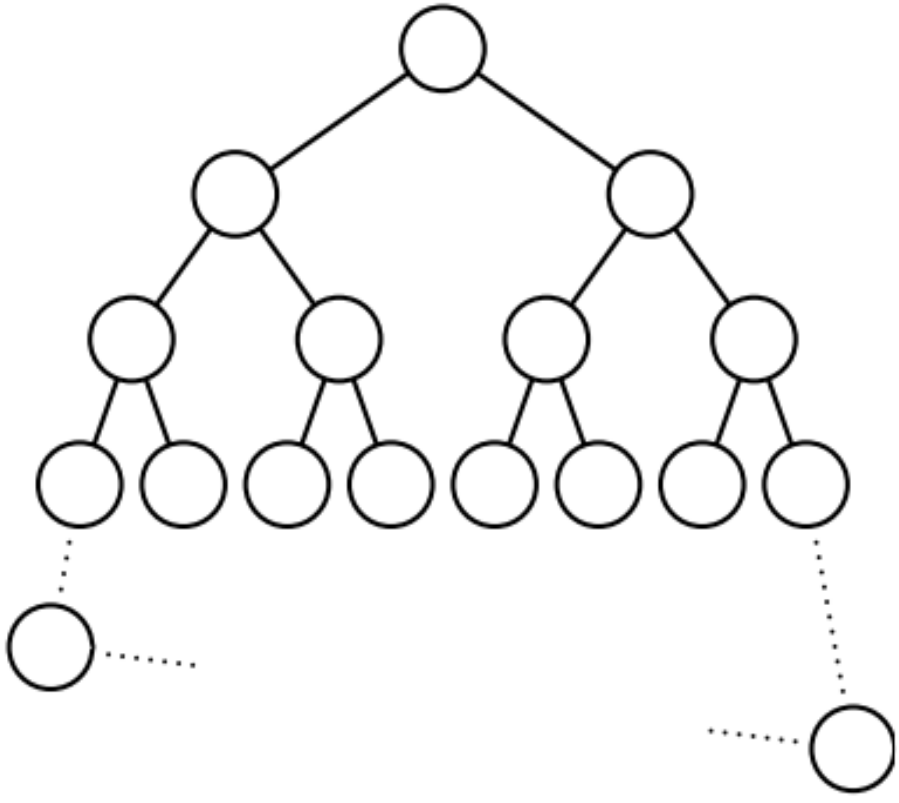
$$T(n) = aT(n/b) + f(n)$$

We therefore cannot use the Master Theorem.

We now show how we can still solve the recursion equation with the recursion tree method.

Example 5

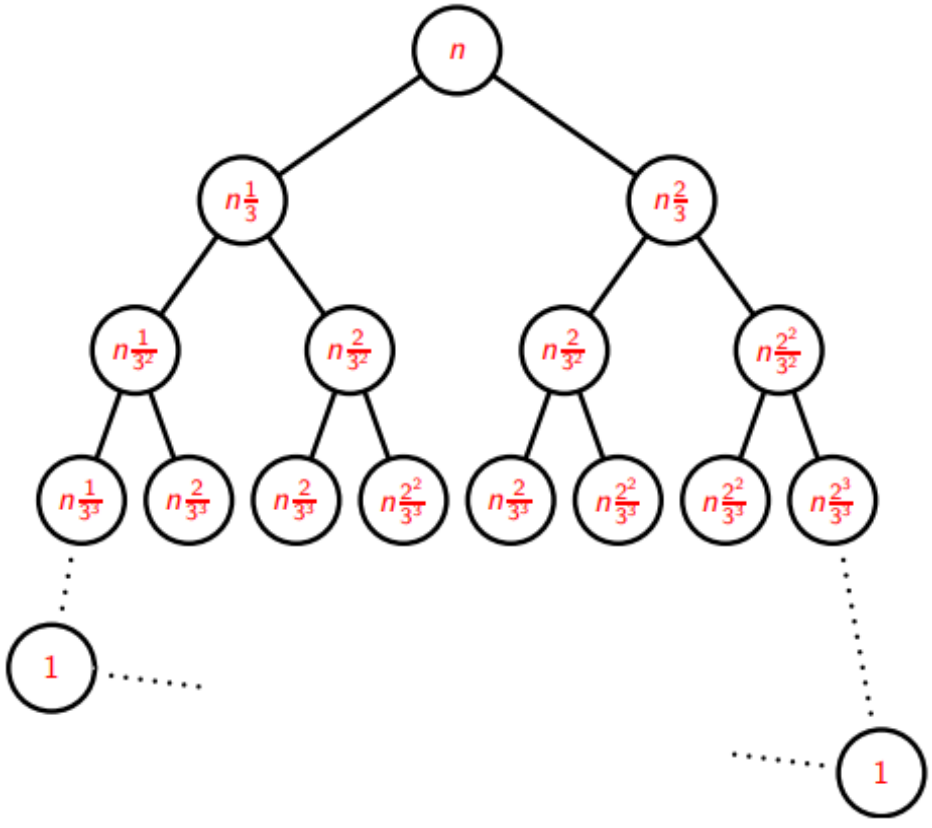
$$T(n) = T(n/3) + T(2n/3) + n$$



Draw tree with 2 (since there are two recursive calls).

Example 5

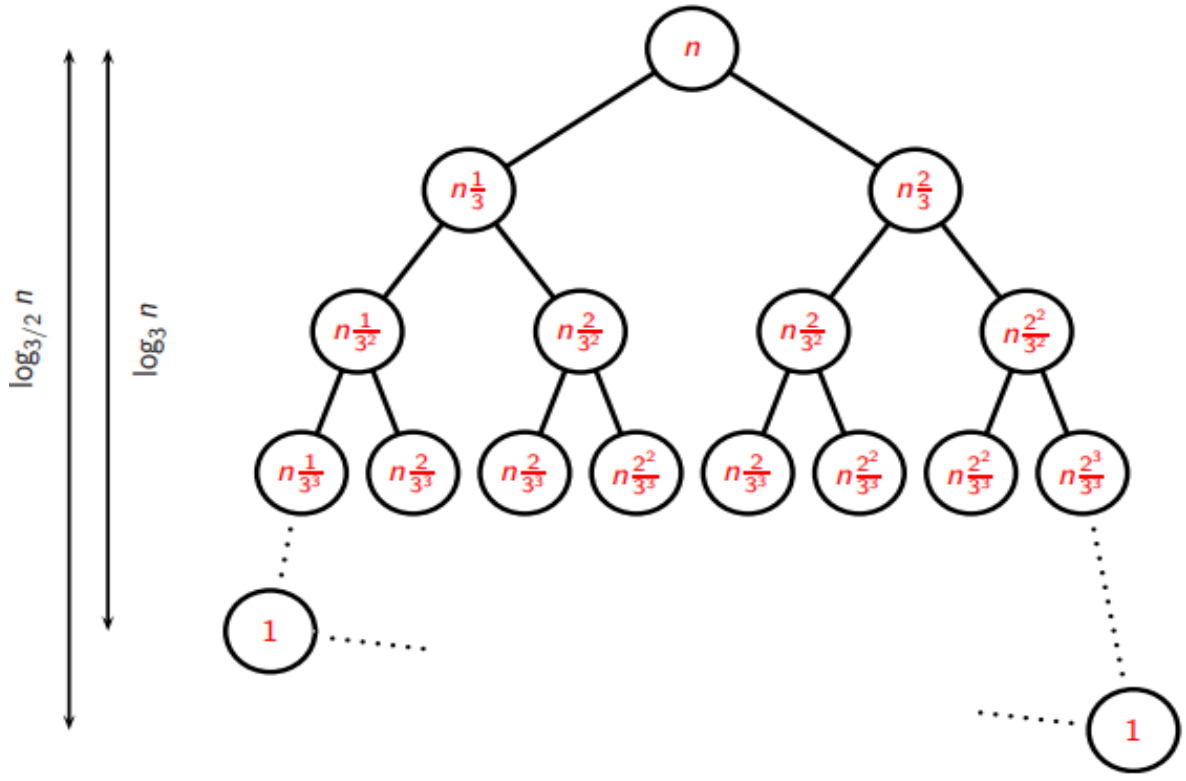
$$T(n) = T(n/3) + T(2n/3) + n$$



Insert input sizes into nodes: multiply by 1/3 (one child) or 2/3 (the other child) when going downwards.

Example 5

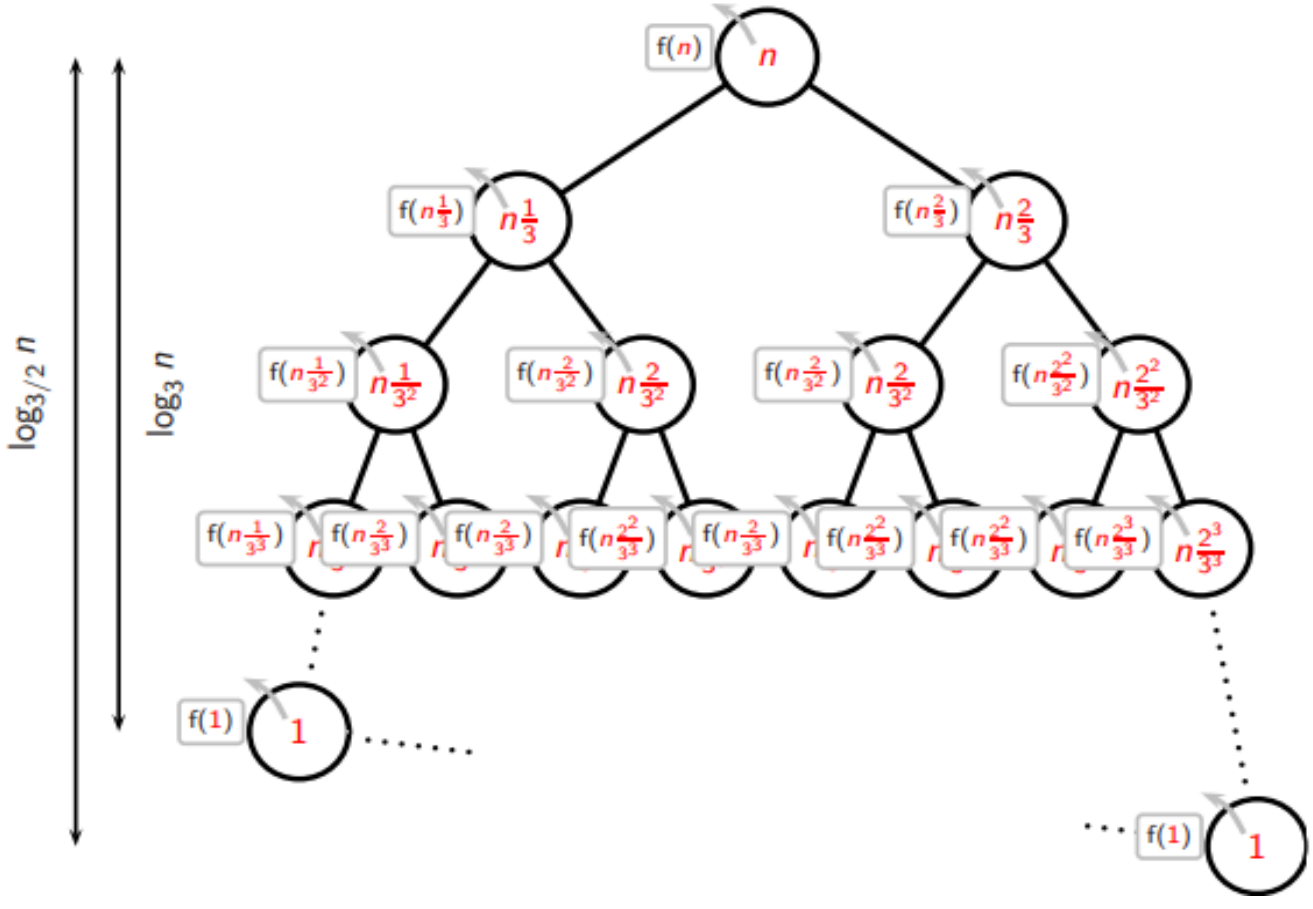
$$T(n) = T(n/3) + T(2n/3) + n$$



Find the longest and shortest path to leaf: $n(1/3)^k = 1 \Leftrightarrow n = 3^k \Leftrightarrow \log_3 n = k$ and $n(2/3)^k = 1 \Leftrightarrow n = (3/2)^k \Leftrightarrow \log_{3/2} n = k$.

Example 5

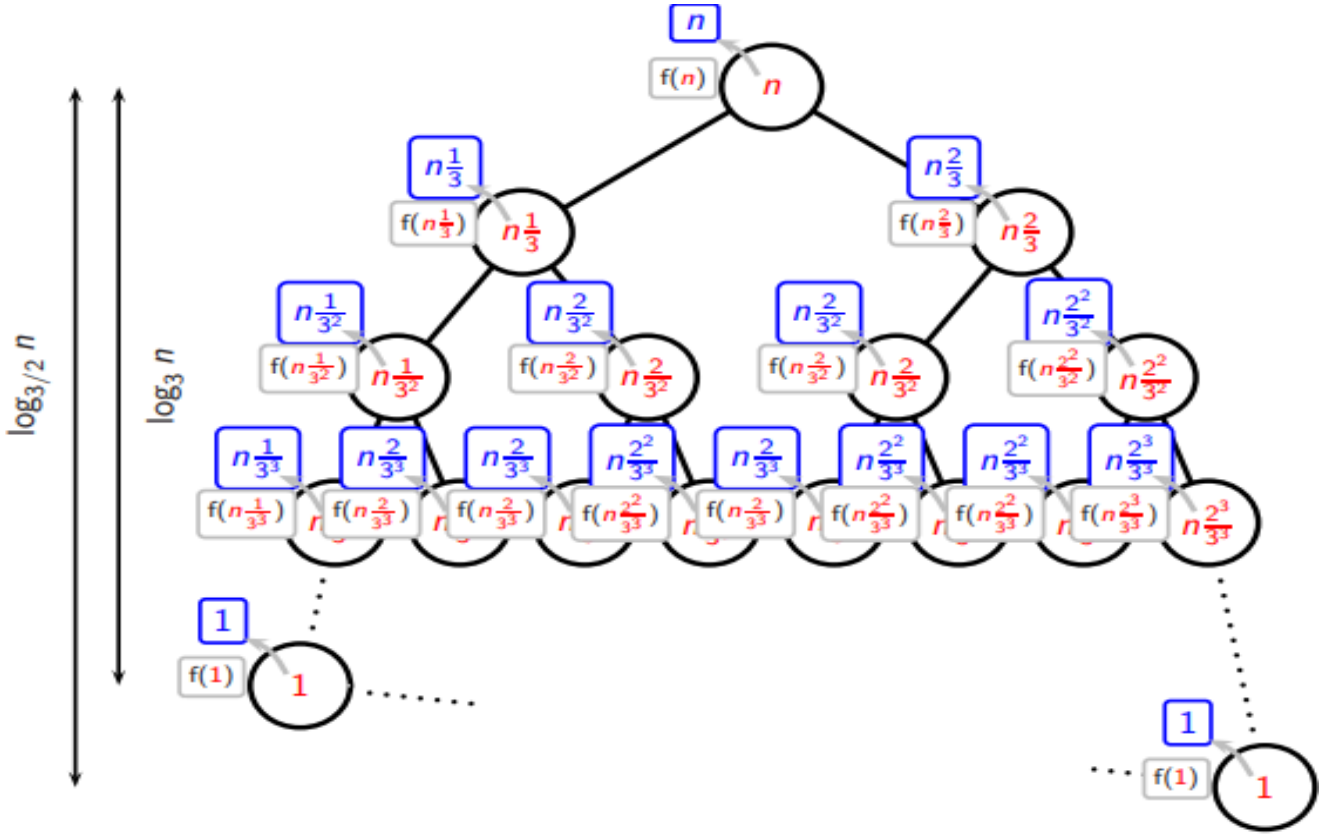
$$T(n) = T(n/3) + T(2n/3) + n$$



Use the function f to convert input size to work.

Example 5

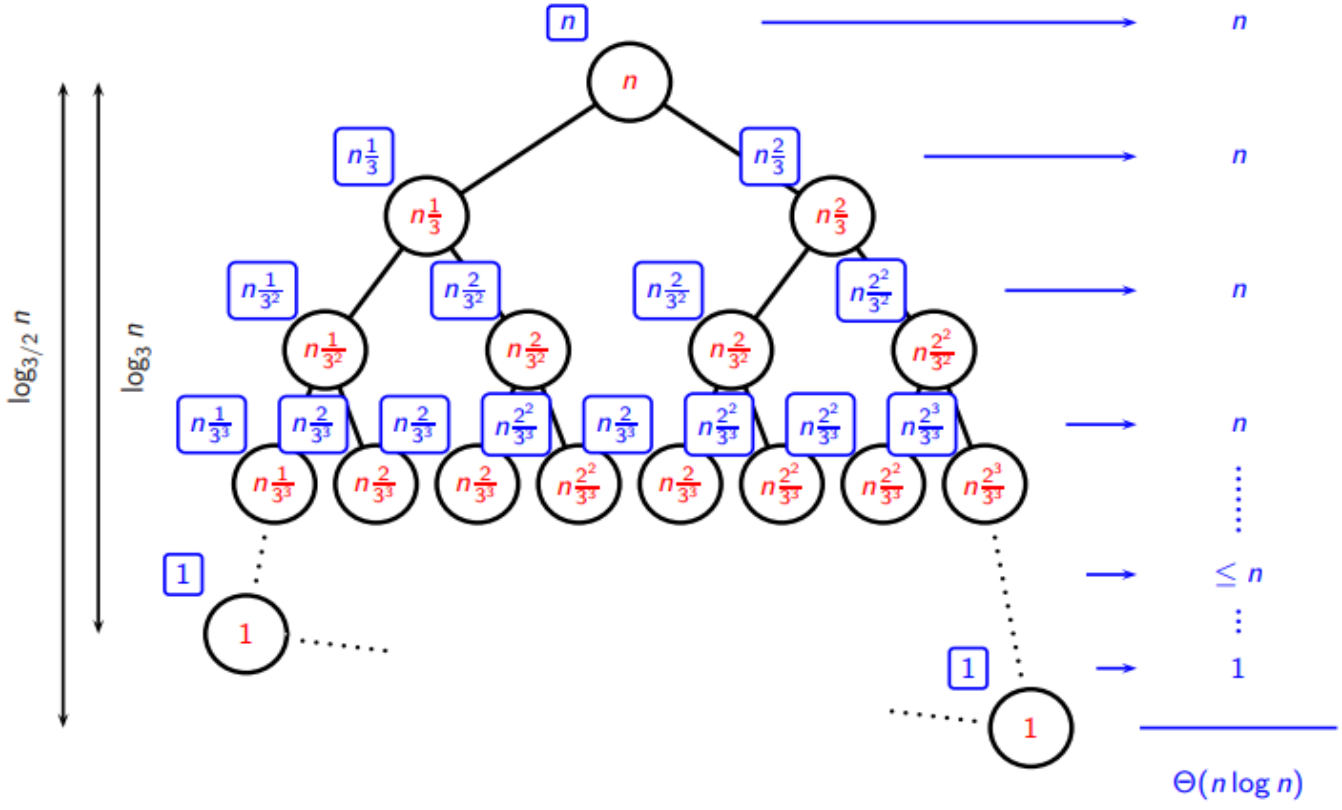
$$T(n) = T(n/3) + T(2n/3) + n$$



Use the function f to convert input size to work.

Example 5

$$T(n) = T(n/3) + T(2n/3) + n$$



Sum each layer and then sum all layers together to find the total work. The answer is Θ(n log n), which we will argue on the next page.

Example 5

A node with input of size x does f (x) = x work. It has two children with inputs of sizes x/3 and 2x/3, which therefore do f(x/3) = x/3 and f(2x/3) = 2x/3 work. Since x/3 + 2x/3 = x, we see that the work of the node is equal to the sum of the work of its children.

It follows that the sum of work in layer k is equal to the sum of the work in layer k + 1, if layer k + 1 is a full layer (all nodes in layer k have two children). So all full layers have the same sum of work. For the root layer, this sum is clearly n, so for all full layers the sum of the work in the layer is equal to n. There are log3 n full layers, so the total work is $\Omega(n \log n)$.[1]

For non-full layers the sum can only be smaller (leaves have no children, so their input size is not carried over to the next layer), i.e. at most n. There are log3/2 n layers in total (full and non-full), so the total work is $O(n \log n)$. Overall, we have shown that the total work is $\Theta(n \log n)$.

---

[1] Here it is used that logarithms with different bases are a constant factor of each other, cf. mathematical fact on page 16.

Floors and ceilings in recursion equations

There is one detail that we haven't talked about so far. We'll use Mergesort as an example.

The recursion equation for Mergesort is written as:

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

But the two recursive calls cannot be exactly the same in size if n is odd. That is, the recursion equation is actually called:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \qquad (2)$$

Does this difference matter to our calculation of running time?

The answer is no, as we shall see.

Floors and ceilings in recursion equations

For a path down the recursion tree, we let n_i denote the size of an input at layer number i (the root layer is set to number 0).

For the recursion equation T(n) = 2T(n/2) + n we have:

$$n_0 = n$$

$$n_1 = n_0/2 = n/2$$

$$n_2 = n_1/2 = (n/2)/2 = n/2^2$$

$$n_3 = n_2/2 = (n/2^2)/2 = n/2^3$$

.

.

.

$$n_i = n/2^i$$

Floors and ceilings in recursion equations

We now look at the recursion equation $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$

Since

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

we get:

$$n_0 = n$$

$$n_1 \leq \lceil n_0/2 \rceil < n_0/2 + 1 = n/2 + 1$$

$$n_2 \leq \lceil n_1/2 \rceil < n_1/2 + 1 < (n/2 + 1)/2 + 1 = n/2^2 + 1/2 + 1$$

$$n_3 \leq \lceil n_2/2 \rceil < n_2/2 + 1 < (n/2^2 + 1/2 + 1)/2 + 1 = n/2^3 + 1/2^2 + 1/2 + 1$$

.

.

.

$$n_i < n/2^i + 1/2^{i-1} + \cdots + 1/2 + 1$$

Floors and ceilings in recursion equations

We now look at the recursion equation $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$

Since

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

and we also get:

$$n_0 = n$$

$$n_1 \geq \lceil n_0/2 \rceil > n_0/2 + 1 = n/2 + 1$$

$$n_2 \geq \lceil n_1/2 \rceil > n_1/2 + 1 > (n/2 + 1)/2 + 1 = n/2^2 + 1/2 + 1$$

$$n_3 \geq \lceil n_2/2 \rceil > n_2/2 + 1 > (n/2^2 + 1/2 + 1)/2 + 1 = n/2^3 + 1/2^2 + 1/2 + 1$$

.

.

.

$$n_i > n/2^i + 1/2^{i-1} + \cdots + 1/2 + 1$$

Floors and ceilings in recursion equations

So for the recursion equation $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$ we have shown

$$n_i < n/2^i + (1/2^{i-1} + \cdots + 1/2 + 1)$$

$$n_i > n/2^i - (1/2^{i-1} + \cdots + 1/2 + 1)$$

The expression in parentheses is equal to $1 + c + c^2 + \cdots + c^{i-1}$ for $c = ½$ and is therefore equal to:

$$\frac{(1/2)^i - 1}{1/2 - 1} = \frac{(1/2)^i - 1}{-1/2} = \frac{1 - (1/2)^i}{1/2} = 2 - (1/2)^{i-1} < 2$$

For the recursion equation $T(n) = 2T(n/2) + n$ we showed:

$$n_i = n/2^i$$

Floors and ceilings in recursion equations

So in the two recursion trees for

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$$

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

The input sizes n_i in the nodes at any given level differ by at most plus/minus 2.

For all the functions f that we have, f(n + 2) = O(f (n)).

> For such functions, we obtain the same asymptotic runtime when analyzing recurrence relations with and without floors/ceilings.

Examples: for f (n) = n applies f(n + 2) = n + 2 = O(n) = O(f (n)). For f (n) = n^2 applies f (n + 2) = $(n + 2)^2 = n^2 + 2n + 4 = O(n^2) = O(f(n))$, for f (n) = log n applies f (n + 2) = log(n + 2) ≤ log 2n = 1 + log n = O(log n) = O(f (n)), and for f(n) = 2^n applies $f(n + 2) = 2^{n+2} = 2^2 \cdot 2^n = 4 \cdot 2^n = O(2^n) = O(f(n))$.