

Plan for today

wrap up of greedy algorithms

quiz

break

discuss quiz

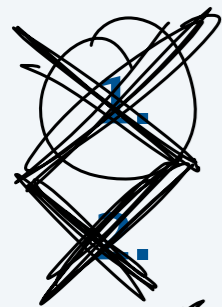
divide + conquer!

2 minutes to think by yourself. Then hold up fingers with your answer.

Recall the job scheduling problem: find largest set of compatible jobs.

Suppose that each job also has a positive weight and the goal is to find a maximum weight subset of mutually compatible intervals.

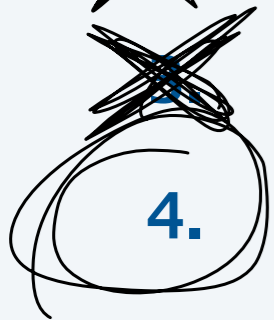
Is the earliest-finish-time-first algorithm still optimal?



Yes, because greedy algorithms are always optimal.

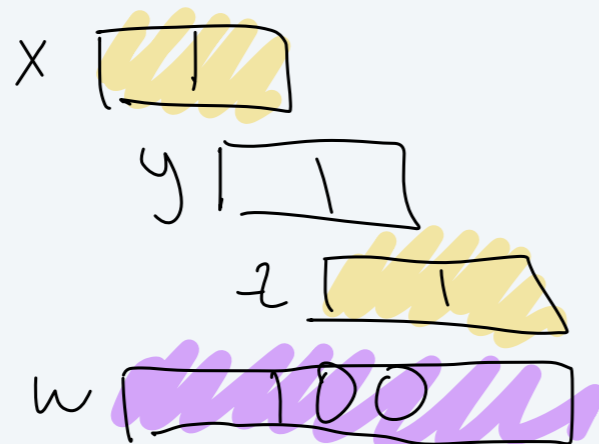
Yes, because the same proof of correctness is valid.

No, because the same proof of correctness is no longer valid.



4.

No, because you could assign a huge weight to a job that overlaps the job with the earliest finish time.



x, z weight: 2
opt weight: 100

Discuss with your table

Suppose that each job also has a positive weight and the goal is to find a maximum weight subset of mutually compatible intervals.

Do you think there is a greedy algorithm for this problem?

Other local criteria?

- largest weight first approach



Greedy algorithms summary

We studied two problems:

Greedy algorithms summary

We studied two problems:

* all-pairs shortest paths

* job scheduling

← Dijkstra's

Greedy algorithms summary

We studied two problems:

- * all-pairs shortest paths
- * job scheduling

Is there a (correct) greedy algorithm for every problem?

Greedy algorithms summary

We studied two problems:

- * all-pairs shortest paths
- * job scheduling

There isn't always a greedy algorithm for a problem.

Greedy algorithms summary

We studied two problems:

* all-pairs shortest paths

* job scheduling

correct
✓

There isn't always a greedy algorithm for a problem.

Discuss with your table: what might you look for in a problem to think it may have a greedy algorithm?

Greedy algorithms summary

We studied two problems:

* all-pairs shortest paths

* job scheduling

correct
✓

There isn't always a greedy algorithm for a problem.

~~✗~~

It takes careful thinking to prove a greedy algorithm works.

Quiz

1. (5 points) Recall the algorithm for the interval scheduling problem. The input to the problem is a set of n jobs with start times s_1, s_2, \dots, s_n and finishing times f_1, f_2, \dots, f_n .

Re-order jobs by finishing time so that $f_1 \leq f_2 \leq \dots \leq f_n$ — $\Theta(n \log n)$

$S = \{\}$

For $j = 1$ to n — n times

 If job j is compatible with the jobs in S : — n

 Add job j to S

Return S

In this problem, you will analyze the runtime of a naive version of this algorithm, and ~~propose an improvement to get a faster runtime~~. You should assume that you can sort a list in $\Theta(n \log n)$ steps in the worst case.

- (a) Suppose that we implement the if statement naively by comparing the new job j to every job in S . For some function $f(n)$ that you choose, give a bound of the form $O(f(n))$ on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).

for loop takes no more than n^2 steps
 $O(n^2)$ overall

$$\Theta(n \log n) \quad \Omega(n \log n)$$

Re-order jobs by finishing time so that $f_1 \leq f_2 \leq \dots \leq f_n$

$S = \{\}$

For $j = 1$ to n — n times

 If job j is compatible with the jobs in S :

 Add job j to S

Return S

\ominus

(b) For this same function $f(n)$, show that the running time of the algorithm on an input of size n is also $\Omega(f(n))$. (This shows an asymptotically tight bound of $\Theta(f(n))$ on the running time.)

$$\frac{n(n-1)}{2} = \Omega(n^2)$$

first time through for loop:	$S = \{\}$	0	
2nd time	:	$ S = 1$	1 +
3rd time	:	$ S = 2$	2 +
	:		+
	:		⋮
n^{th} time	:	$ S = n-1$	$n-1$

2. (10 points) This problem is about the same setup from the homework:

Every fall, your family harvests apples at the old family orchard. You have many varied sizes of bins that you harvest the apples into, and a large fleet of identical carts that can all take a total weight of W . Your family has always done it exactly this way:

- Park a cart at the orchard.
- As bins of apples are finished, pack them onto the cart in exactly the order they arrive.
- Once a bin arrives that would put the cart over its maximum weight, send the cart off and start loading the next cart.

According to this system, we could index the bins with the variable i and write the weight of the i^{th} bin w_i . Because your family members pick apples at different speeds and fill the bins to different levels, the w_i values are varied and unpredictable, and because the apple orchard produces at different levels every year, the total number of bins that will be loaded varies year to year as well.

(a) (3 points) Suppose that you get bins 1, 2, ... 10 with weights 30, 50, 30, 30, 20, 40, 10, 30, 20, 70 and your carts have a capacity of ~~20~~. Is the solution

$\begin{matrix} c1 & c2 & c3 & c4 \\ \hline 30, 50 & 30, 30, 20 & 40, 10, 30, 20 & 70 \end{matrix}$

Cart capacity 100

- Cart 1: bins 1 and 2 (for a total of 80 pounds)
- Cart 2: bins 3 and 4 (for a total of 60 pounds)
- Cart 3: bins 5, 6, 7, and 8 (for a total of 100 pounds)
- Cart 4: bins 9, 10 (for a total of 90 pounds)

yes

optimal?

(b) (3 points) Is the solution from (a) what would be found by the greedy algorithm?

(c) (4 points) In order to prove that the greedy algorithm always uses the fewest carts possible, we can prove that the greedy algorithm “stays ahead” of any optimal solution. Suppose that we identify bins j_1, j_2, \dots, j_m as the final bins that are packed into each cart in some optimal solution, and bins $\ell_1, \ell_2, \dots, \ell_k$ as the final bins packed into each cart by the greedy solution. In the example solution from (a), $j_1 = 2, j_2 = 4, j_3 = 8,$ and $j_4 = 10$.

To prove that the greedy algorithm stays ahead, we would want to prove that (fill in the blank):

For all $r \leq k$, we have $\ell_r \geq \underline{j_r}$.

\uparrow
greedy

Divide-and-conquer paradigm

Divide-and-conquer paradigm

Divide-and-conquer.

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide up problem into several subproblems (of the same kind).

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide up problem into several subproblems (of the same kind).
- Solve (conquer) each subproblem recursively.

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide up problem into several subproblems (of the same kind).
- Solve (conquer) each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide up problem into several subproblems (of the same kind).
- Solve (conquer) each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide up problem into several subproblems (of the same kind).
- Solve (conquer) each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide up problem into several subproblems (of the same kind).
- Solve (conquer) each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Our goals.

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide up problem into several subproblems (of the same kind).
- Solve (conquer) each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Our goals.

- Design correct algorithms using this powerful algorithm design strategy

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide up problem into several subproblems (of the same kind).
- Solve (conquer) each subproblem recursively.
- Combine solutions to subproblems into overall solution.

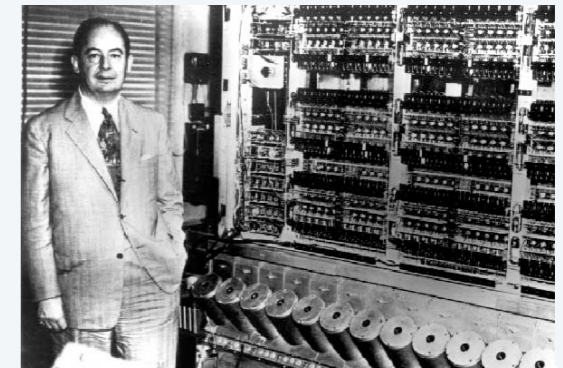
Our goals.

- Design correct algorithms using this powerful algorithm design strategy
- Be able to analyze the runtimes of recursive algorithms.

Mergesort

input

A	L	G	O	R	I	T	H	M	S
---	---	---	---	---	---	---	---	---	---



**First Draft
of a
Report on the
EDVAC**

John von Neumann

Mergesort

- Recursively sort left half.

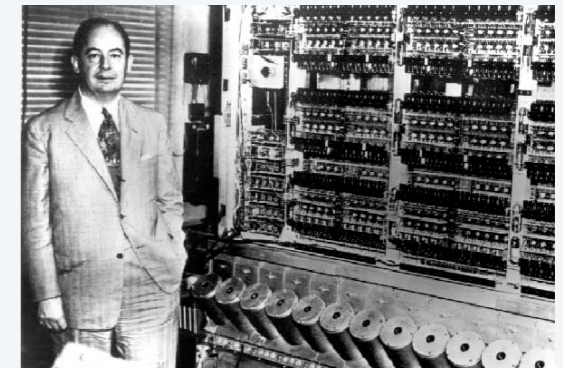
input

A	L	G	O	R	I	T	H	M	S
---	---	---	---	---	---	---	---	---	---

(A bracket is drawn under the first five letters: A, L, G, O, R.)

sort left half

A	G	L	O	R	I	T	H	M	S
---	---	---	---	---	---	---	---	---	---



**First Draft
of a
Report on the
EDVAC**

John von Neumann

Mergesort

- Recursively sort left half.
- Recursively sort right half.

input

A	L	G	O	R	I	T	H	M	S
---	---	---	---	---	---	---	---	---	---

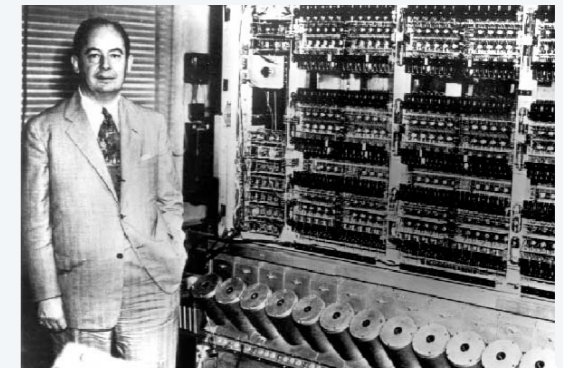
sort left half

A	G	L	O	R	I	T	H	M	S
---	---	---	---	---	---	---	---	---	---

sort right half

A	G	L	O	R	H	I	M	S	T
---	---	---	---	---	---	---	---	---	---

(A hand-drawn arrow points from the 'H' in the second array back to the 'I' in the first array, indicating a swap.)



**First Draft
of a
Report on the
EDVAC**

John von Neumann

Mergesort

- Recursively sort left half.
- Recursively sort right half.
- Merge two halves to make sorted whole.

input

A	L	G	O	R	I	T	H	M	S
---	---	---	---	---	---	---	---	---	---

sort left half

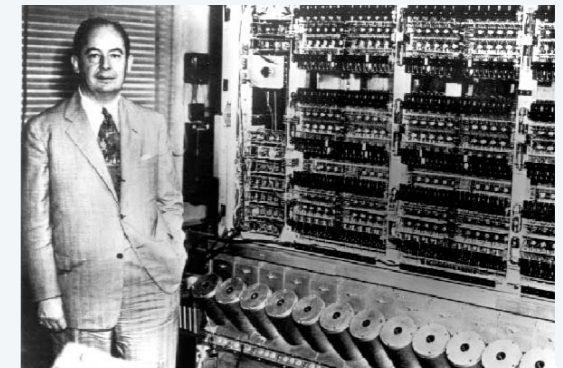
A	G	L	O	R	I	T	H	M	S
---	---	---	---	---	---	---	---	---	---

sort right half

A	G	L	O	R	H	I	M	S	T
---	---	---	---	---	---	---	---	---	---

merge results

A	G	H	I	L	M	O	R	S	T
---	---	---	---	---	---	---	---	---	---



**First Draft
of a
Report on the
EDVAC**

John von Neumann

Mergesort

Mergesort

- Recursively sort left half.

Mergesort

- Recursively sort left half.
- Recursively sort right half.

Mergesort

- Recursively sort left half.
- Recursively sort right half.
- Merge two halves to make sorted whole.

mergesort(L):

$L_1 =$ first half of L } n time

$L_2 =$ second half of L

sorted- $L_1 =$ mergesort(L_1)

sorted- $L_2 =$ mergesort(L_2)

return merged sorted- L_1 and sorted- L_2

Merging

Merging

Goal. Combine two sorted lists A and B into a sorted whole C .

Merging

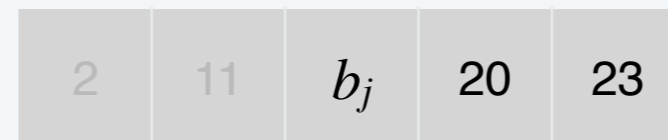
Goal. Combine two sorted lists A and B into a sorted whole C .

- Scan A and B from left to right.
- Compare a_i and b_j .
- If $a_i \leq b_j$, append a_i to C (no larger than any remaining element in B).
- If $a_i > b_j$, append b_j to C (smaller than every remaining element in A).

sorted list A



sorted list B



merge to form sorted list C



Merge demo

Given two sorted lists A and B , merge into sorted list C .

sorted list A

3	7	10	14	18
---	---	----	----	----

sorted list B

2	11	16	20	23
---	----	----	----	----

Merge demo

Given two sorted lists A and B , merge into sorted list C .

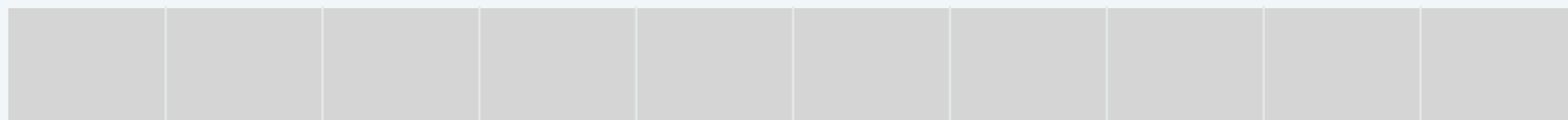
sorted list A



sorted list B



sorted list C



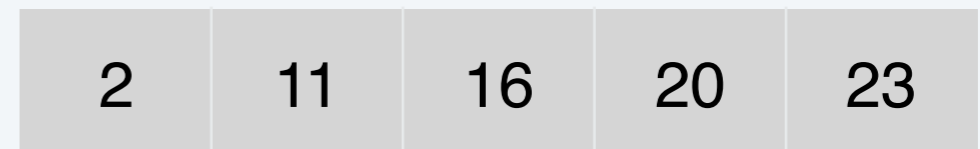
Merge demo

Given two sorted lists A and B , merge into sorted list C .

sorted list A



sorted list B



compare minimum entry in each list: copy 2

sorted list C



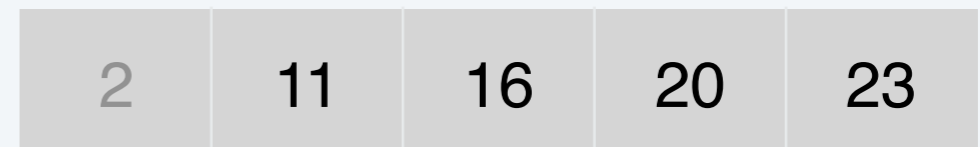
Merge demo

Given two sorted lists A and B , merge into sorted list C .

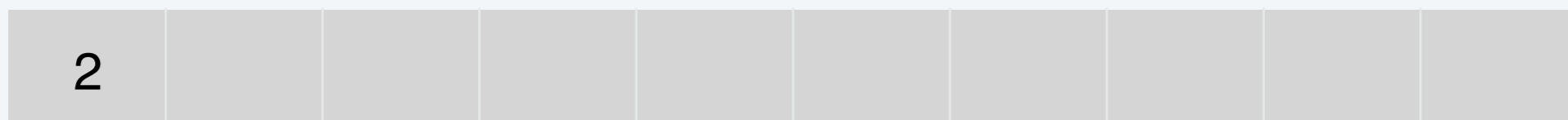
sorted list A



sorted list B



sorted list C



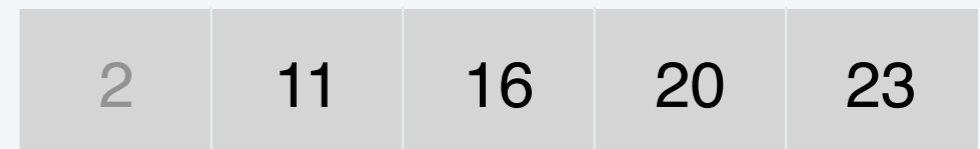
Merge demo

Given two sorted lists A and B , merge into sorted list C .

sorted list A

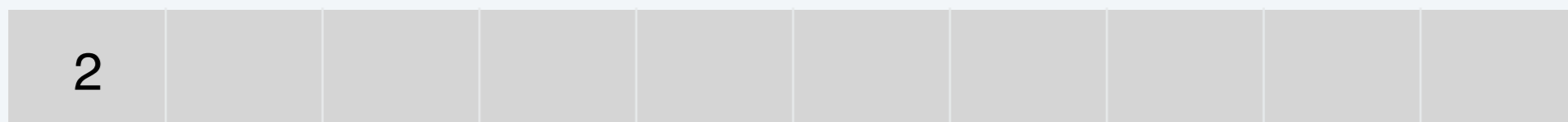


sorted list B



compare minimum entry in each list: copy 3

sorted list C



Merge demo

Given two sorted lists A and B , merge into sorted list C .

sorted list A



sorted list B



sorted list C



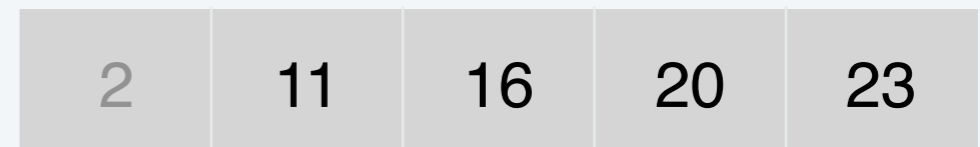
Merge demo

Given two sorted lists A and B , merge into sorted list C .

sorted list A



sorted list B



compare minimum entry in each list: copy 7

sorted list C



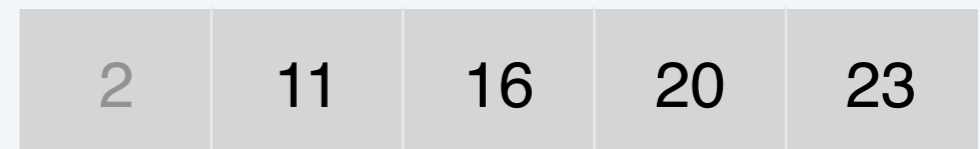
Merge demo

Given two sorted lists A and B , merge into sorted list C .

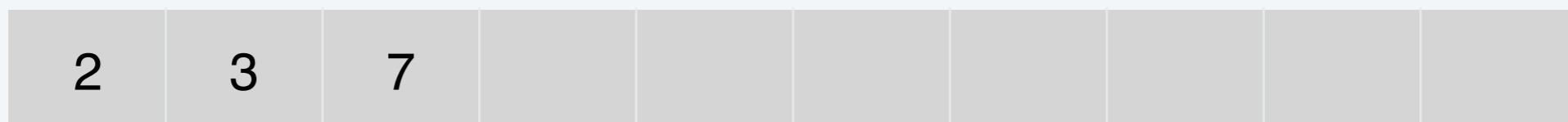
sorted list A



sorted list B



sorted list C



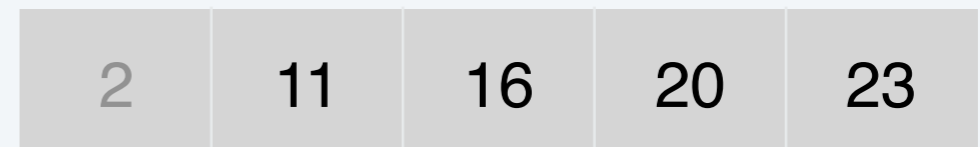
Merge demo

Given two sorted lists A and B , merge into sorted list C .

sorted list A



sorted list B



compare minimum entry in each list: copy 10

sorted list C



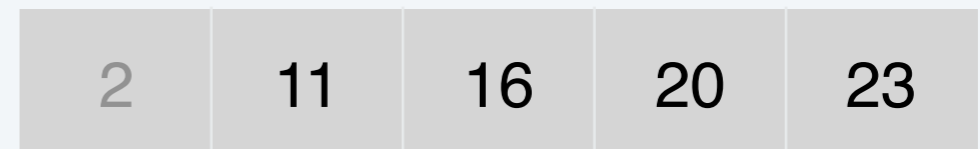
Merge demo

Given two sorted lists A and B , merge into sorted list C .

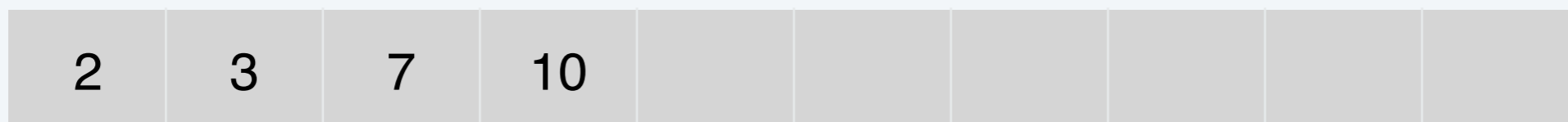
sorted list A



sorted list B



sorted list C



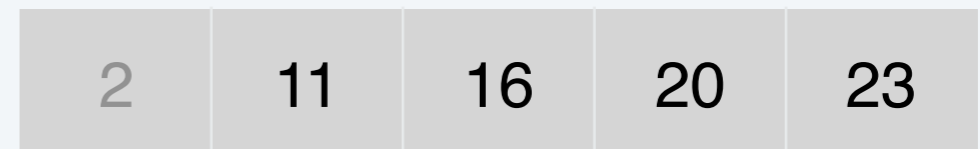
Merge demo

Given two sorted lists A and B , merge into sorted list C .

sorted list A

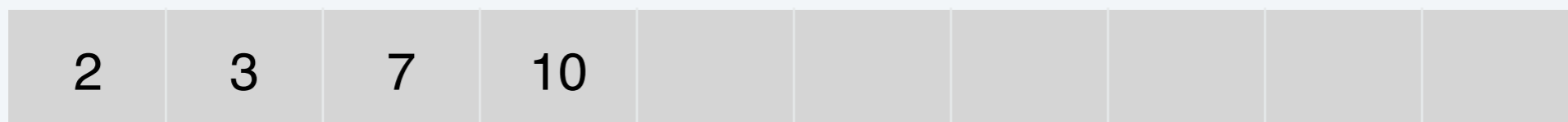


sorted list B



compare minimum entry in each list: copy 11

sorted list C



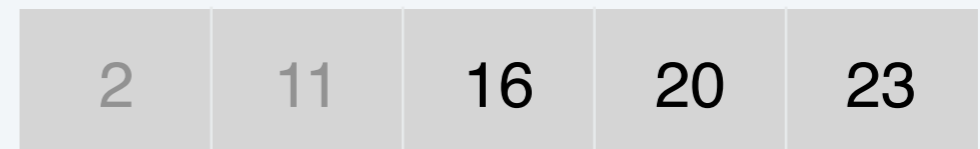
Merge demo

Given two sorted lists A and B , merge into sorted list C .

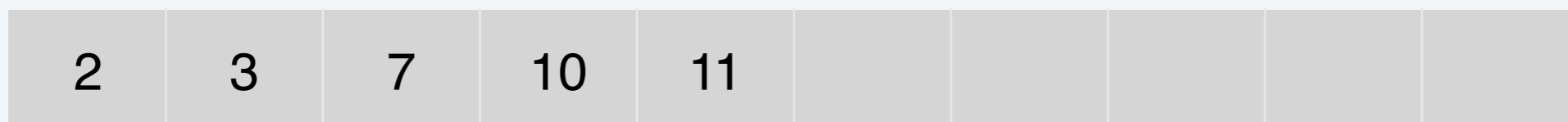
sorted list A



sorted list B



sorted list C



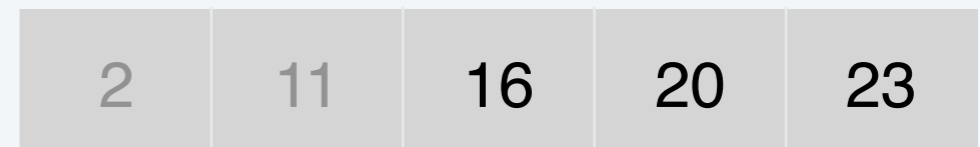
Merge demo

Given two sorted lists A and B , merge into sorted list C .

sorted list A

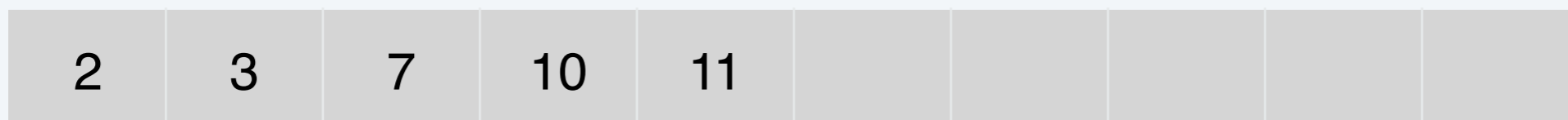


sorted list B



compare minimum entry in each list: copy 14

sorted list C



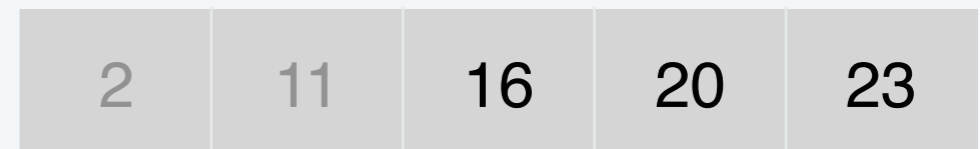
Merge demo

Given two sorted lists A and B , merge into sorted list C .

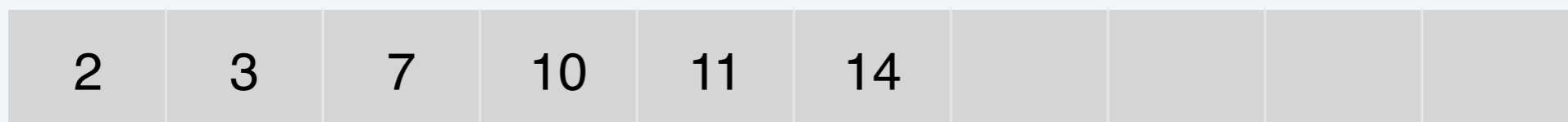
sorted list A



sorted list B



sorted list C



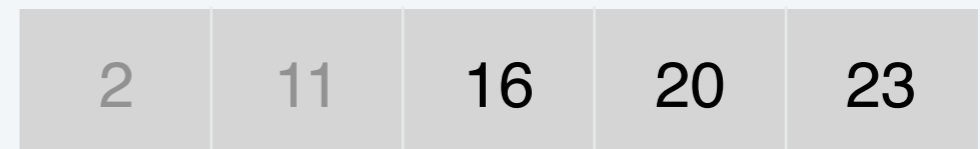
Merge demo

Given two sorted lists A and B , merge into sorted list C .

sorted list A



sorted list B



compare minimum entry in each list: copy 16

sorted list C



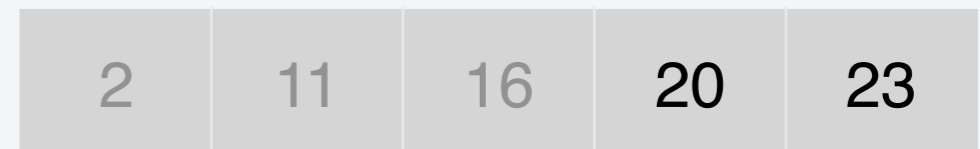
Merge demo

Given two sorted lists A and B , merge into sorted list C .

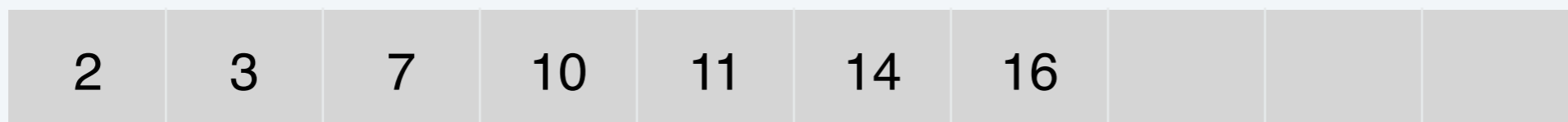
sorted list A



sorted list B



sorted list C



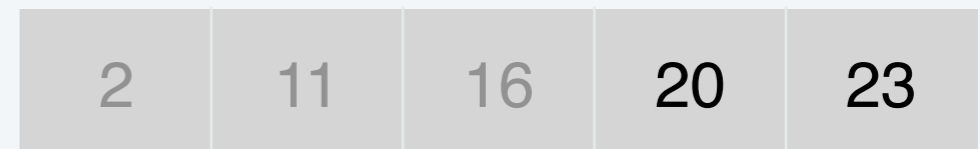
Merge demo

Given two sorted lists A and B , merge into sorted list C .

sorted list A

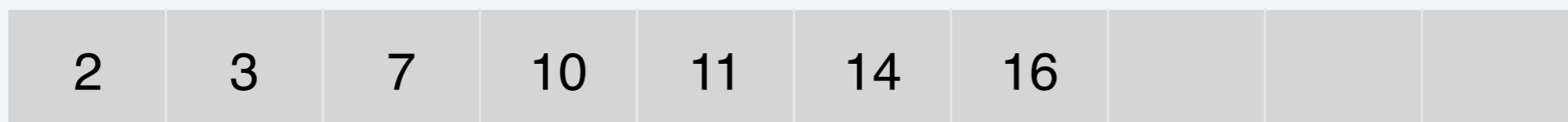


sorted list B



compare minimum entry in each list: copy 18

sorted list C



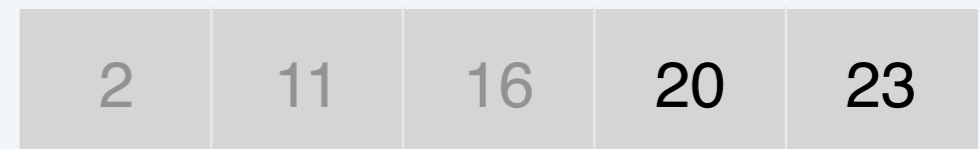
Merge demo

Given two sorted lists A and B , merge into sorted list C .

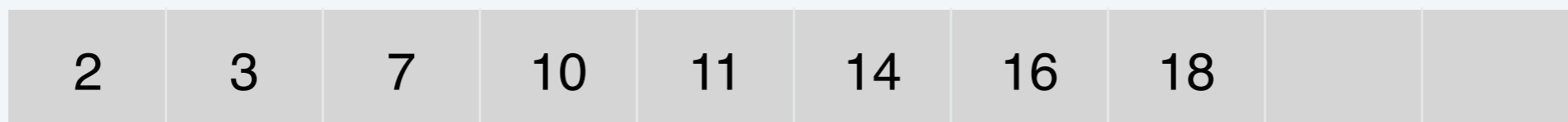
sorted list A



sorted list B



sorted list C



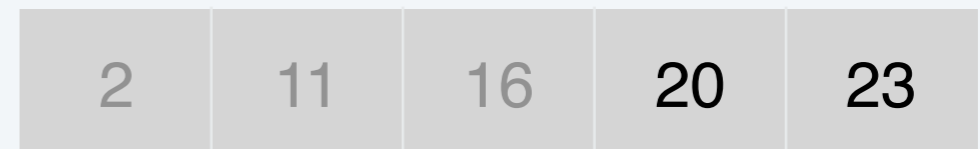
Merge demo

Given two sorted lists A and B , merge into sorted list C .

sorted list A

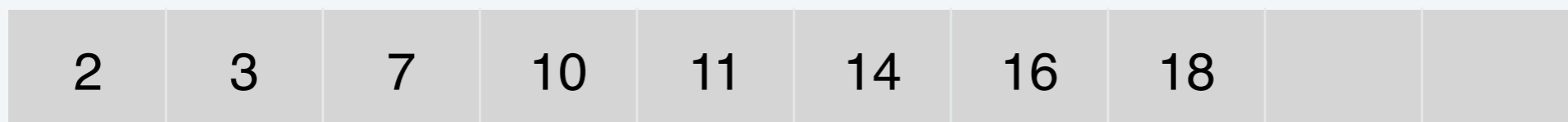


sorted list B



list A exhausted: copy 20

sorted list C



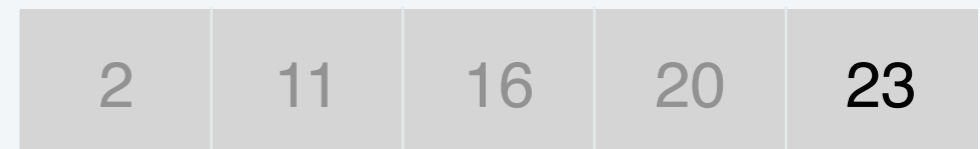
Merge demo

Given two sorted lists A and B , merge into sorted list C .

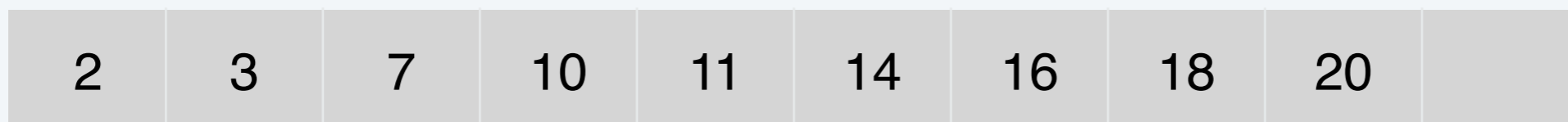
sorted list A



sorted list B



sorted list C



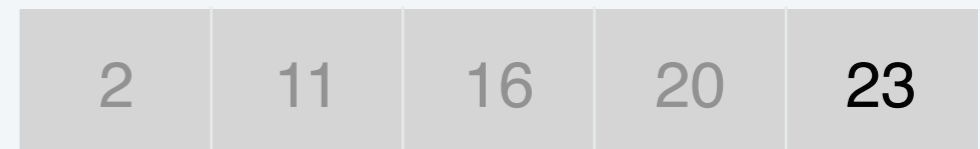
Merge demo

Given two sorted lists A and B , merge into sorted list C .

sorted list A

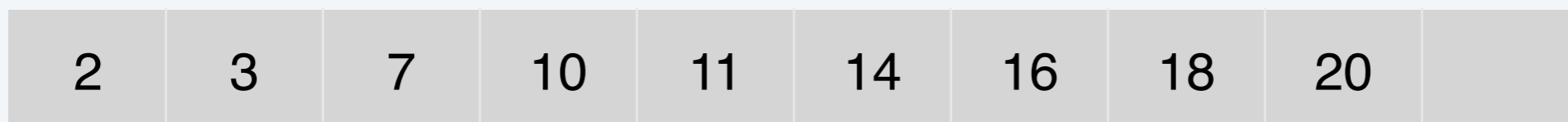


sorted list B



list A exhausted: copy 23

sorted list C



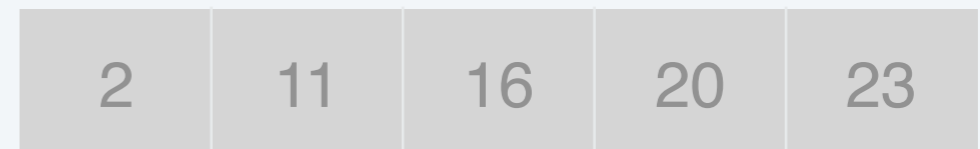
Merge demo

Given two sorted lists A and B , merge into sorted list C .

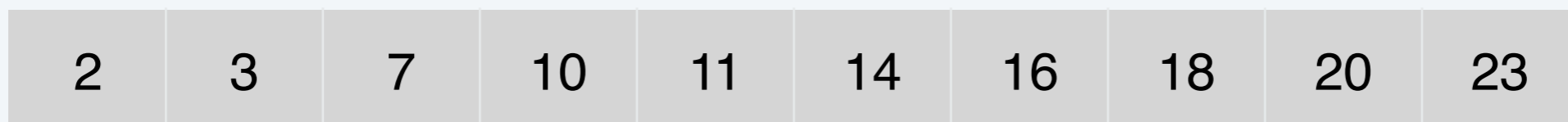
sorted list A



sorted list B



sorted list C



Merge demo

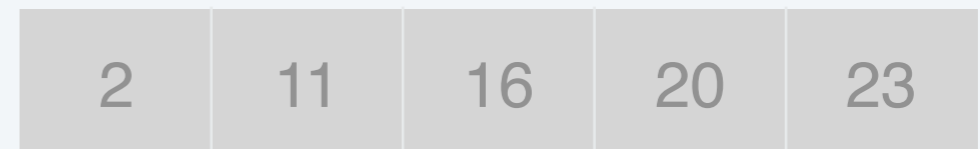
Is there a difference between a worst-case and best-case input for number of steps taken by this algorithm?

1. Yes
2. No

sorted list A



sorted list B



sorted list C



Merge demo

Is there a difference between a worst-case and best-case input for number of steps taken by this algorithm? Discuss with table

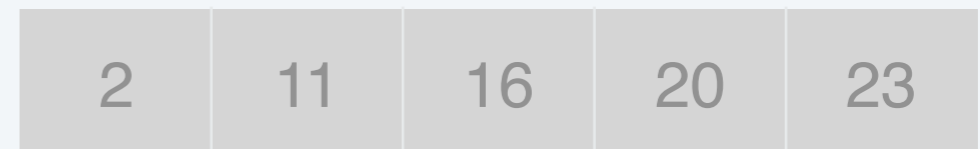
1. Yes

2. No

sorted list A



sorted list B



sorted list C



Merge demo

Is there a difference between a worst-case and best-case input for number of steps taken by this algorithm? Discuss with table

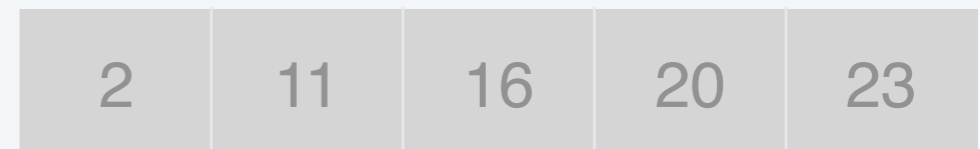
1. Yes

2. No

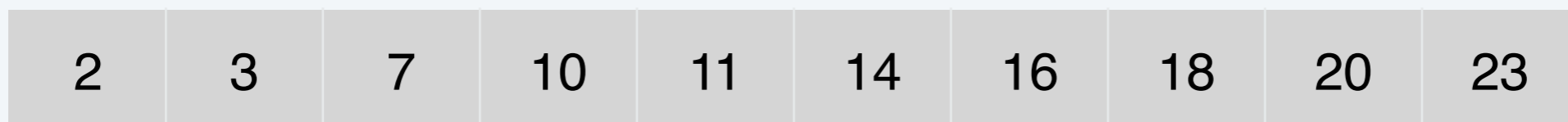
sorted list A



sorted list B



sorted list C



Merge demo

Is there a difference between a worst-case and best-case input for number of steps taken by this algorithm? Discuss with table: what is the runtime?

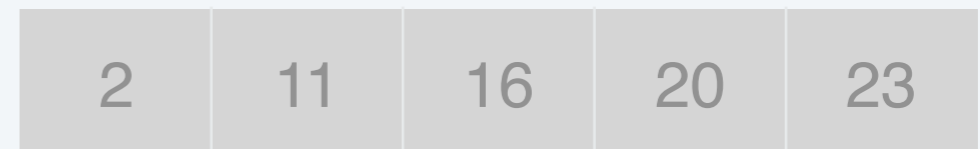
1. Yes

2. No

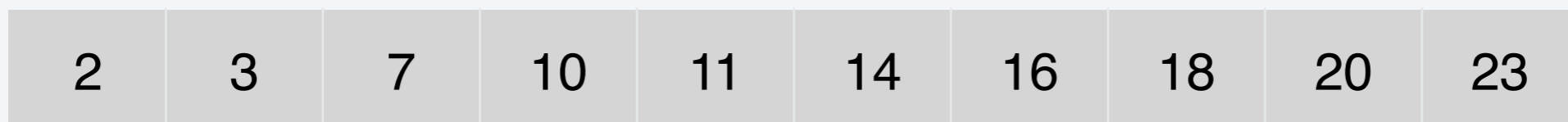
sorted list A



sorted list B



sorted list C



Mergesort

- Recursively sort left half.
- Recursively sort right half.
- Merge two halves to make sorted whole.

mergesort(L) :

L₁ = first half of L

L₂ = second half of L

sorted_L₁ = mergesort(L₁)

sorted_L₂ = mergesort(L₂)

return merged L₁ and L₂

Approach # 1: unroll the recurrence

Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n)$ is...


$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

assuming n
is a power of 2




Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

assuming n
is a power of 2



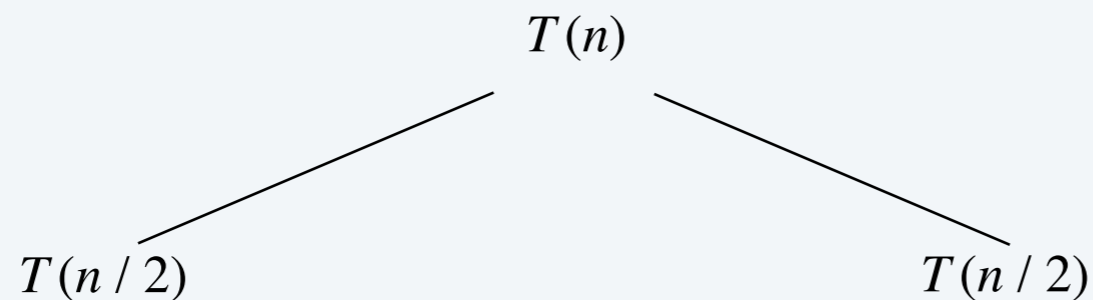

$T(n)$

Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

assuming n
is a power of 2

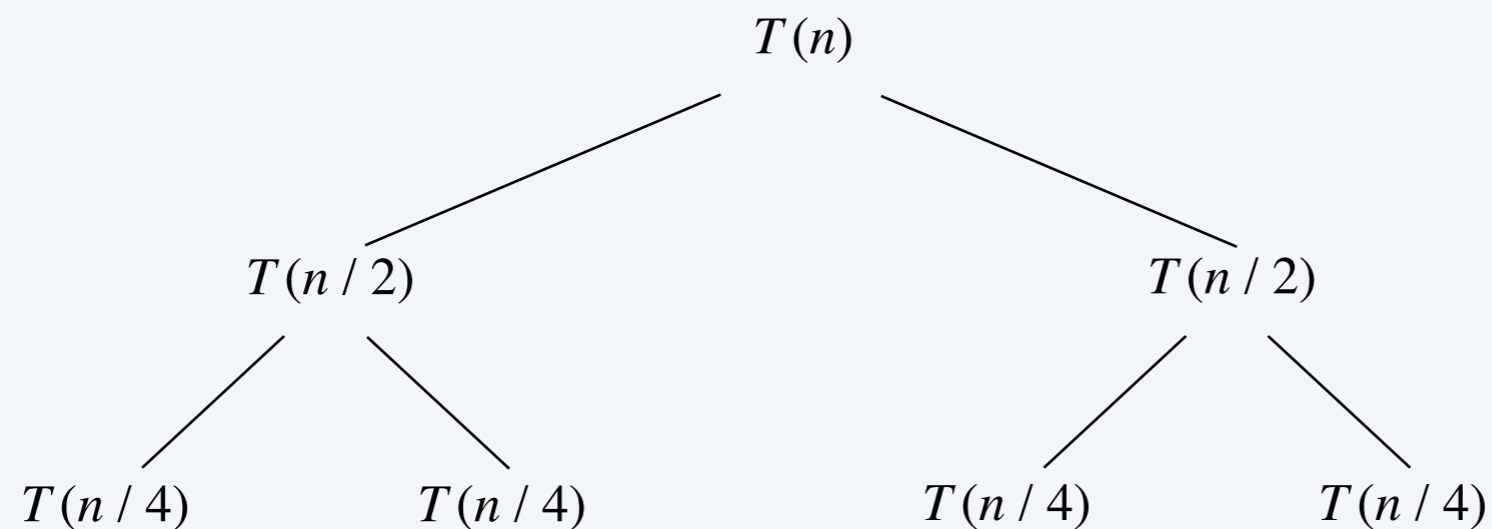


Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

assuming n
is a power of 2

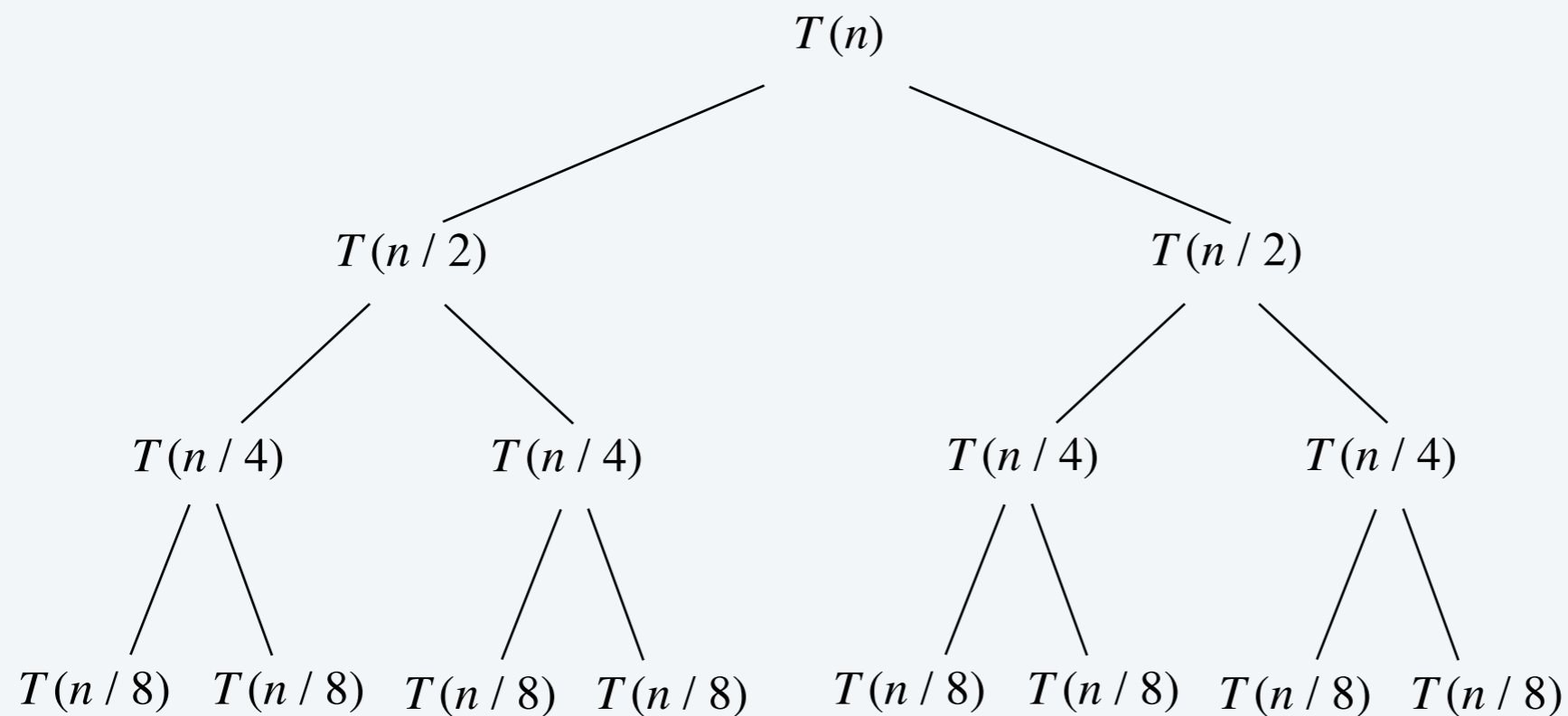


Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

assuming n
is a power of 2

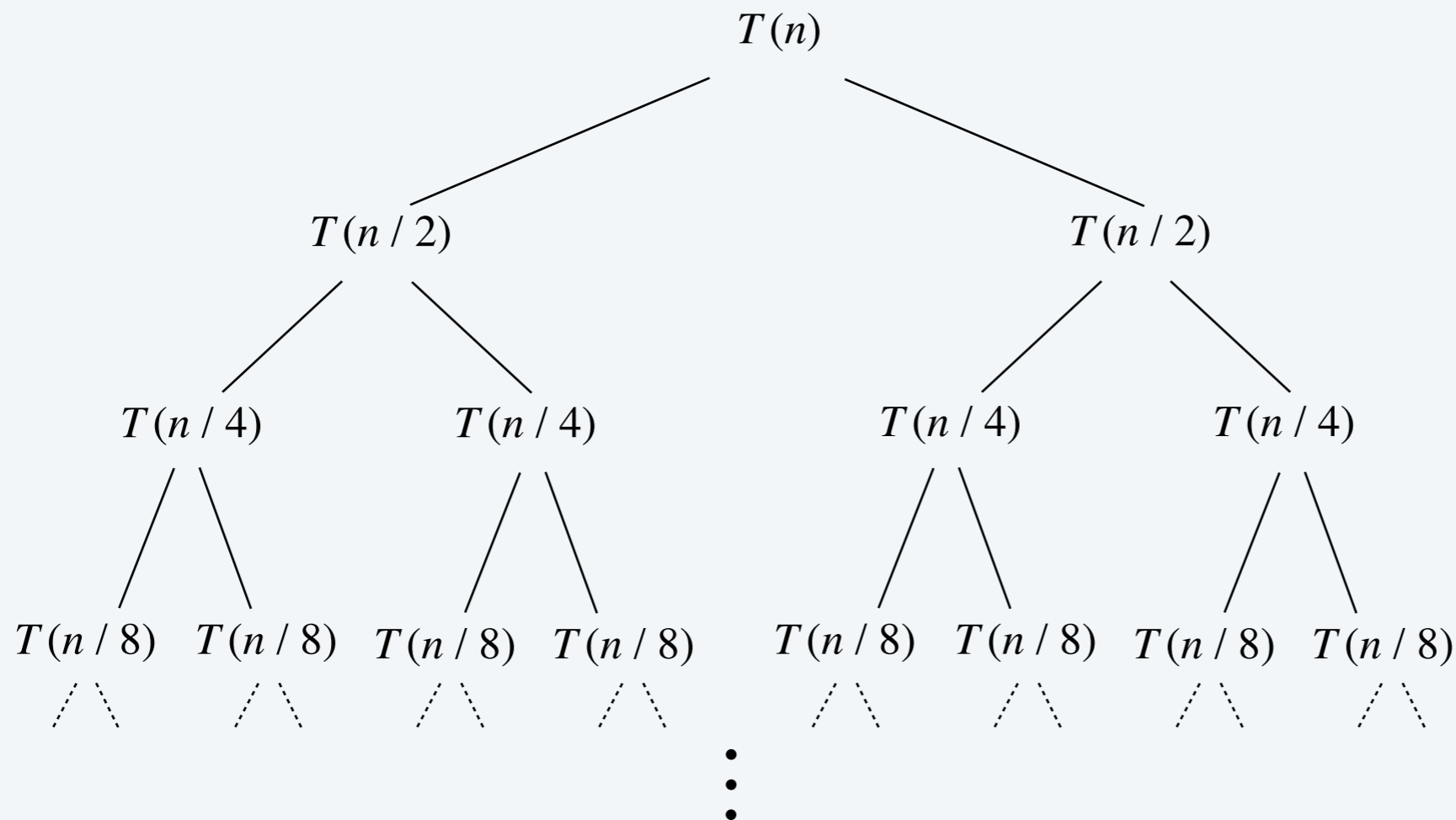


Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

assuming n
is a power of 2

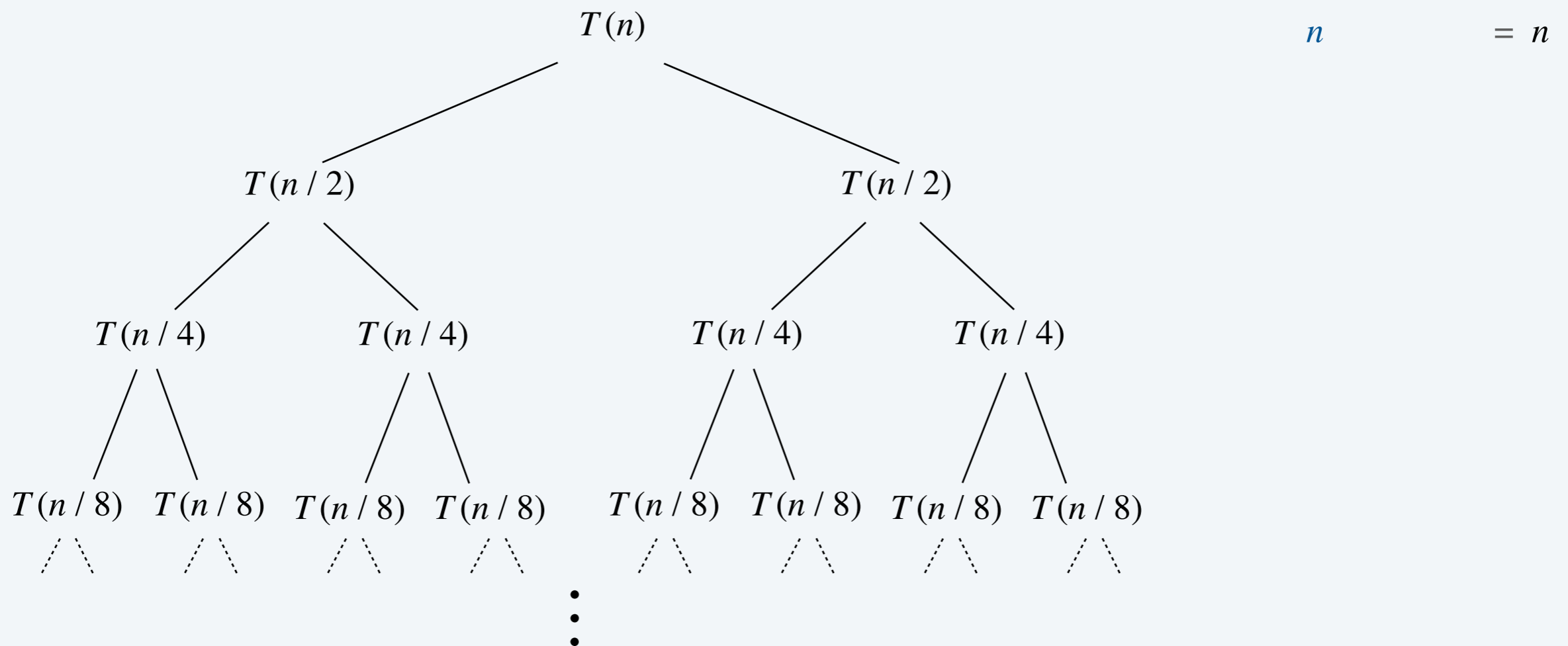


Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

assuming n
is a power of 2

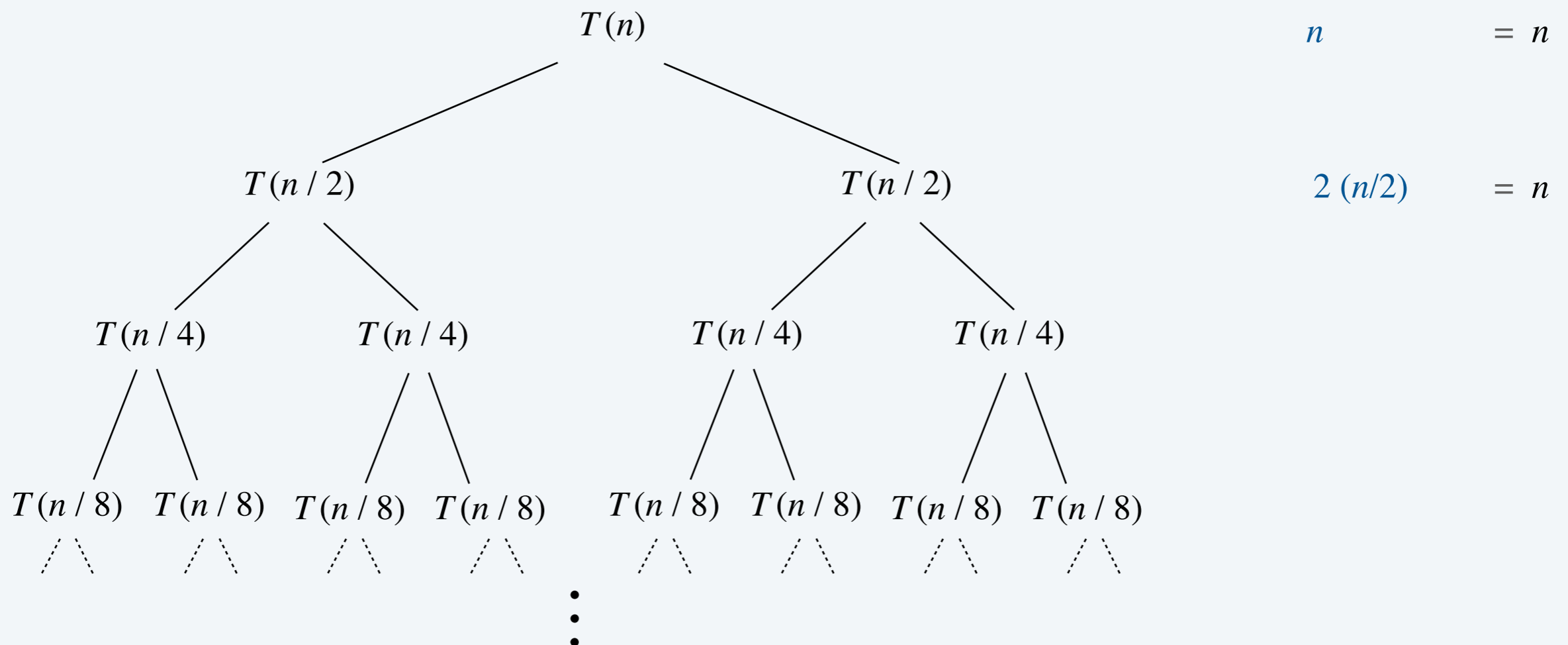


Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

assuming n
is a power of 2

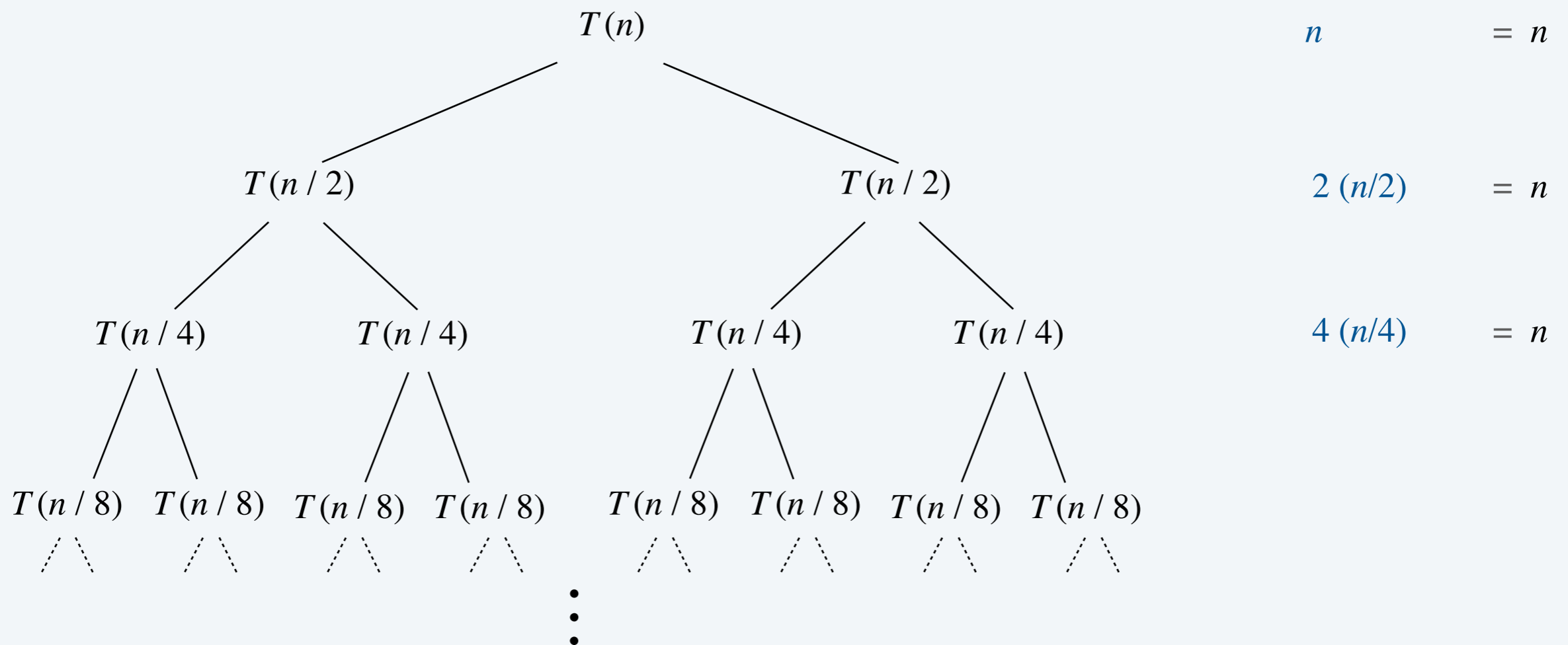


Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

assuming n
is a power of 2

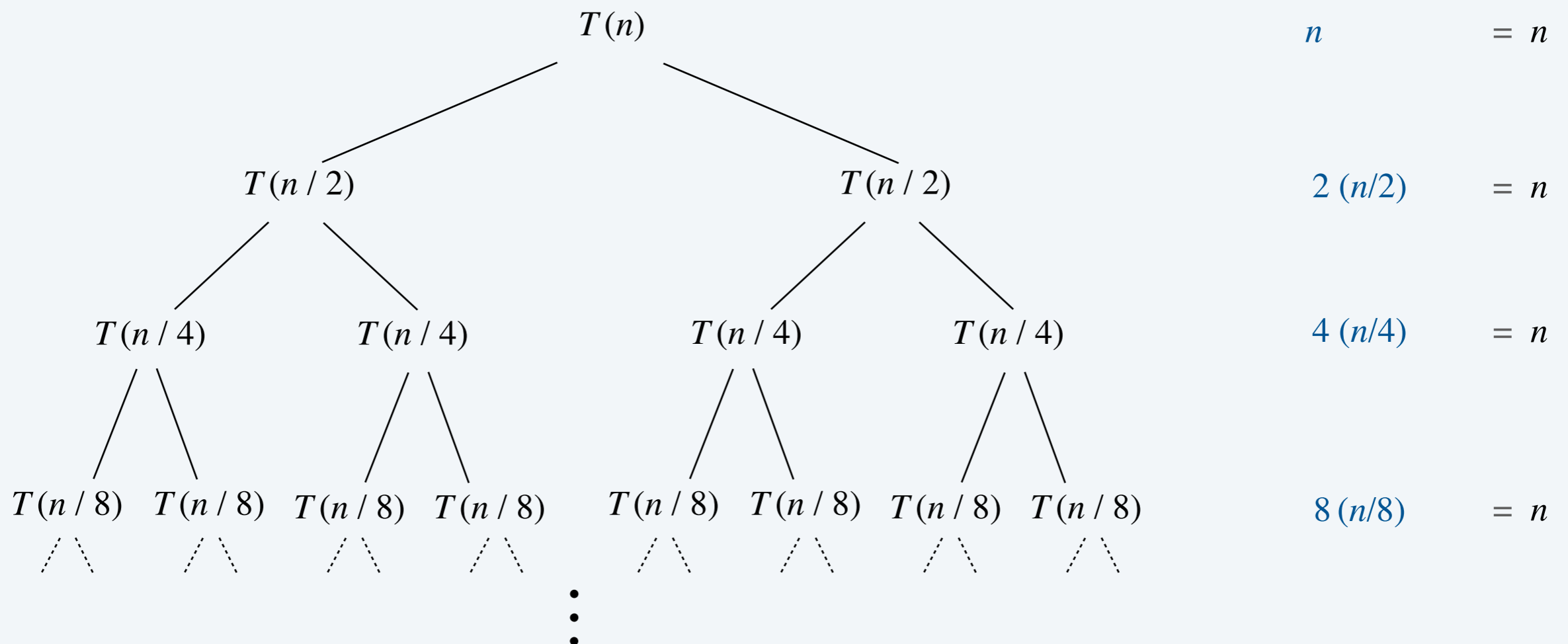


Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

assuming n
is a power of 2

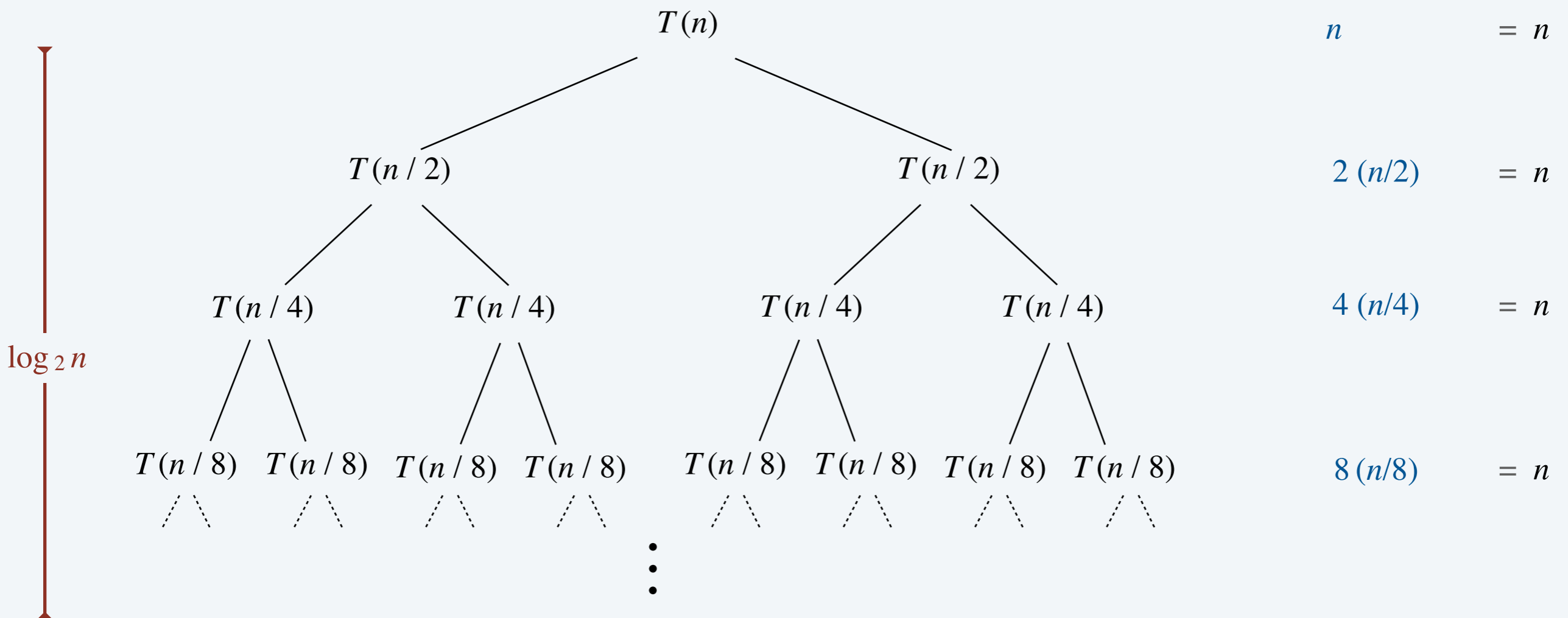


Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

assuming n
is a power of 2

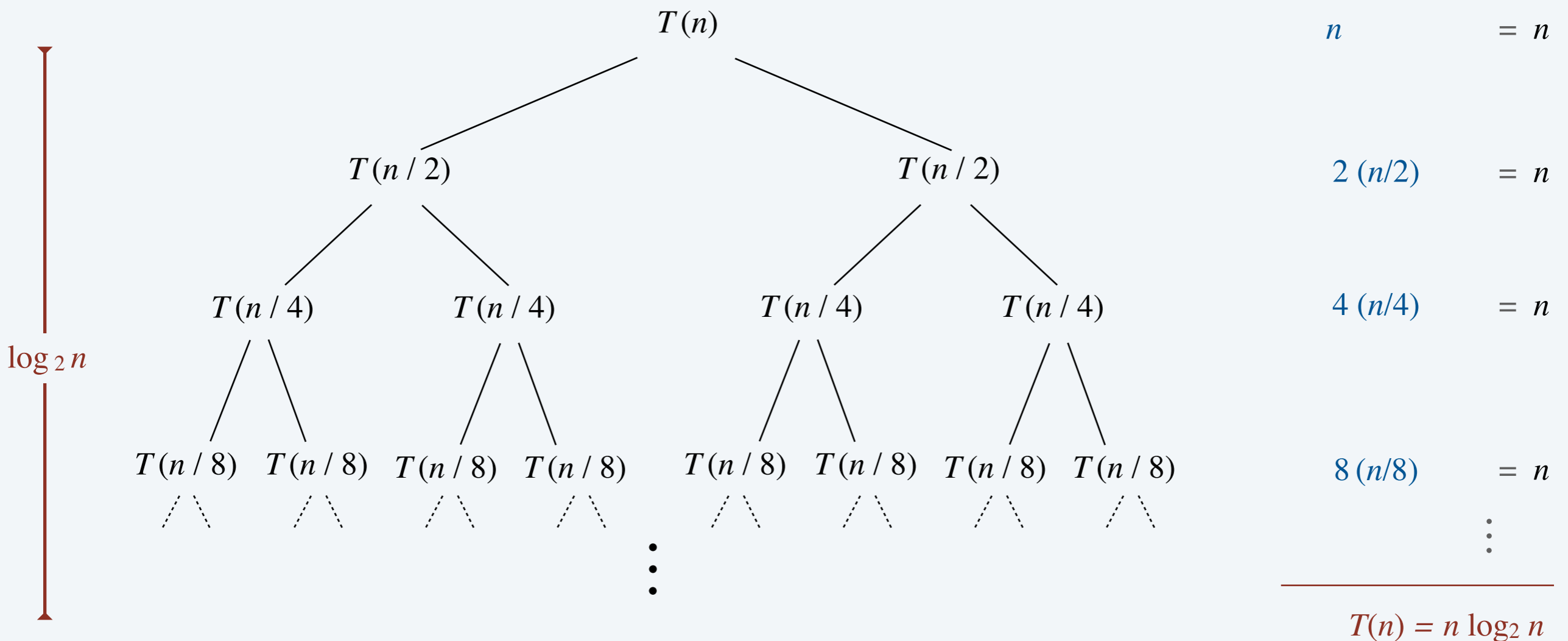


Approach # 1: unroll the recurrence

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

assuming n
is a power of 2



Choose an answer

fancymergesort(L):

L_1 = first third of L

L_2 = second third of L

L_3 = last third of L

$sorted_L_1$ = mergesort(L_1)

$sorted_L_2$ = mergesort(L_2)

$sorted_L_3$ = mergesort(L_3)

return merged L_1, L_2, L_3

What is a valid recurrence relation for fancymergesort?

1. $T(n) = n^2$
2. $T(n) = 3T(n/3) + n$
3. $T(n) = cn \log_3 n$
4. $T(n) = nT(n) + 3n$