Algorithms and Data Structures

Upper and Lower Bounds for Sorting, Matrix Multiplication

Worst-case Running Times, Upper Bounds

Algorithm **Mergesort**(**A**[$i,\ldots,j$])

If $i=j$, return i

$q=(i+j)/2$

$\mathbf{A}_{left}$=**Mergesort**(**A**[$i,\ldots,q$])
$\mathbf{A}_{right}$=**Mergesort**(**A**[$q+1,\ldots,n$])
return **Merge**($\mathbf{A}_{left}$, $\mathbf{A}_{right}$)

Recurrence relation:

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

where $f(n) = O(n)$

If we solve the recurrence relation we obtain
$T(n) = O(n \lg n)$

To be exactly precise

Recurrence relation: For some constant c,

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn, & \text{when } n > 2. \\ T(2) \leq c, & \text{otherwise.} \end{cases}$$

How do we solve this recurrence relation?

How do we solve this recurrence relation?

How do we solve this recurrence relation?

The Master Theorem

The Master Theorem is a very general theorem for solving recurrence relations.

The Master Theorem

The Master Theorem is a very general theorem for solving recurrence relations.

Suppose $T(n) \leq \alpha T\left(\lceil n/b \rceil\right) + O(n^d)$
for some constants $\alpha > 0,\ b > 1$ and $d \geq 0$.

$$\text{Then,}\ \ T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b \alpha \\ O(n^d \log_b n), & \text{if } d = \log_b \alpha \\ O(n^{\log_b \alpha}), & \text{if } d < \log_b \alpha \end{cases}$$

The Master Theorem

The Master Theorem is a very general theorem for solving recurrence relations.

Suppose $T(n) \leq \alpha T\left(\lceil n/b \rceil\right) + O(n^d)$

for some constants $\alpha > 0,\ b > 1$ and $d \geq 0$.

$$\text{Then,}\quad T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b \alpha \\ O(n^d \log_b n), & \text{if } d = \log_b \alpha \\ O(n^{\log_b \alpha}), & \text{if } d < \log_b \alpha \end{cases}$$

Example: For **MergeSort**, $\alpha = b = 2$ and $d = 1$, we get $O(n \log n)$

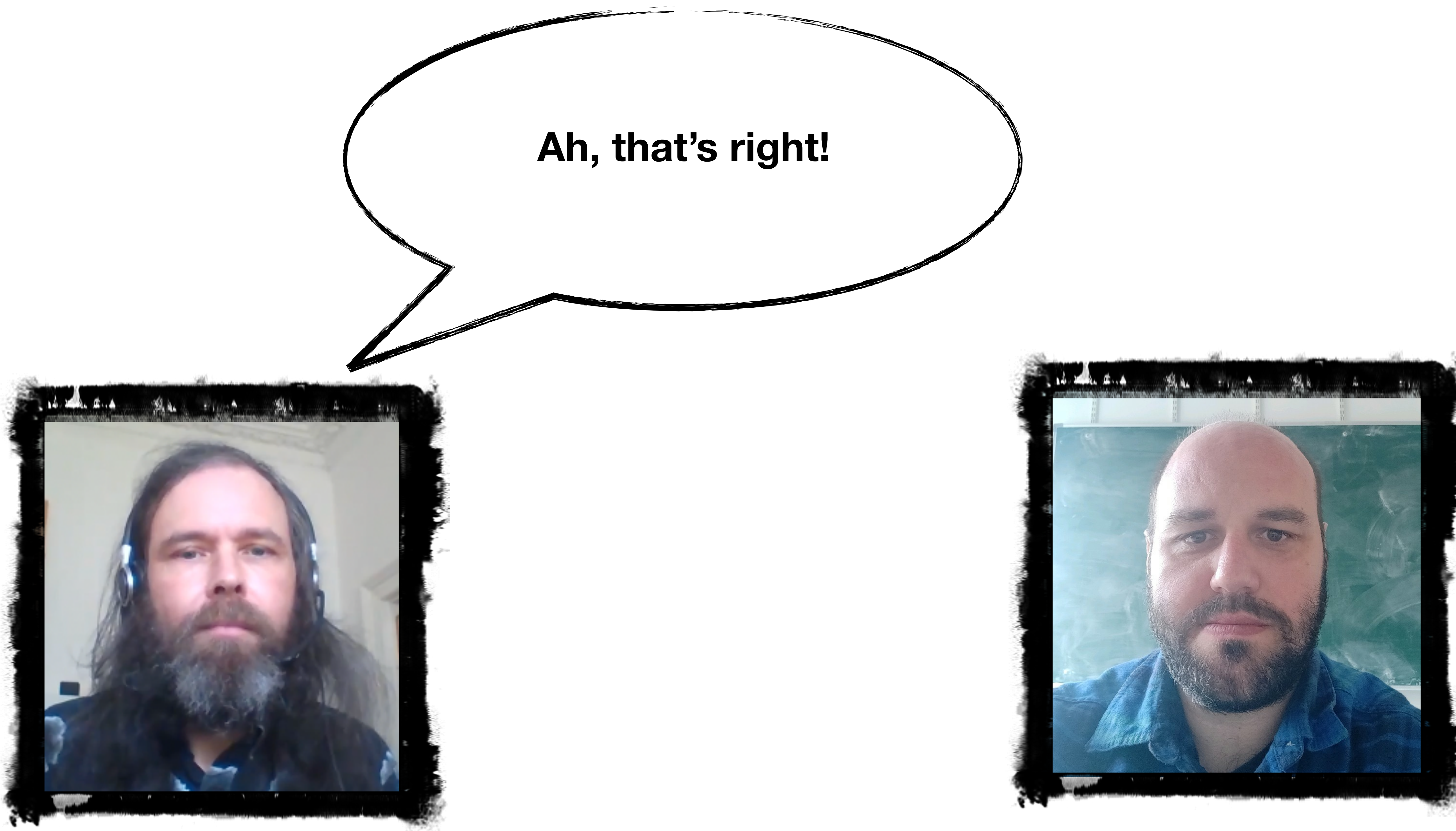
How do we solve this recurrence relation?

How do we solve this recurrence relation?

How do we solve this recurrence relation?

How do we solve this recurrence relation?

How do we solve this recurrence relation?



How do we solve recurrence relations?

How do we solve recurrence relations?

Two main techniques:

How do we solve recurrence relations?

Two main techniques:

"Guess and verify": We guess the solution, and verify that it works after substituting in the recurrence relation. Often the argument is by *induction on n.*

How do we solve recurrence relations?

Two main techniques:

"Guess and verify": We guess the solution, and verify that it works after substituting in the recurrence relation. Often the argument is by *induction on n.*

"Unrolling the recursion": Figure out the solution for the first few *levels*, and then identifying a pattern. In the end we sum the solutions over all the levels. Often also called the method of "recursion trees".

Let's use the "guess and verify" technique on this recurrence relation

Recurrence relation: For some constant c,

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn, & \text{when } n > 2. \\ T(2) \leq c, & \text{otherwise.} \end{cases}$$

Let's use the "guess and verify" technique on this recurrence relation

Recurrence relation: For some constant c,

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn, & \text{when } n > 2. \\ T(2) \leq c, & \text{otherwise.} \end{cases}$$

Let's consider the simpler version:

$$T(n) = \begin{cases} 2T(n/2) + cn, & \text{when } n > 2. \\ T(2) \leq c, & \text{otherwise.} \end{cases}$$

Let's use the "guess and verify" technique on this recurrence relation

Let's use the "guess and verify" technique on this recurrence relation

For simplicity, assume $n = 2^k$ for some integer k.

Let’s use the “guess and verify” technique on this recurrence relation

For simplicity, assume $n = 2^k$ for some integer k.

We guess that: $T(n) \leq cn \log_2 n$ for all $n \geq 2$.

Let's use the "guess and verify" technique on this recurrence relation

For simplicity, assume $n = 2^k$ for some integer k.

We guess that: $T(n) \leq cn \log_2 n$ for all $n \geq 2$.

It remains to prove that our guess was correct.

Let's use the "guess and verify" technique on this recurrence relation

For simplicity, assume $n = 2^k$ for some integer k.

We guess that: $T(n) \leq cn \log_2 n$ for all $n \geq 2$.

It remains to prove that our guess was correct.

For $n = 2$, we have $T(2) \leq c \leq 2c = c \cdot 2 \log_2 2$.

Let's use the "guess and verify" technique on this recurrence relation

For simplicity, assume $n = 2^k$ for some integer k.

We guess that: $T(n) \leq cn \log_2 n$ for all $n \geq 2$.

It remains to prove that our guess was correct.

For $n = 2$, we have $T(2) \leq c \leq 2c = c \cdot 2 \log_2 2$.

For $n = 4$, we have $T(4) = 2T(2) + 4c \leq 2c + 4c = 6c$.
$4c \log_2 4 = 8c > 6c$.

Let's use the "guess and verify" technique on this recurrence relation

For simplicity, assume $n = 2^k$ for some integer k.

We guess that: $T(n) \leq cn \log_2 n$ for all $n \geq 2$.

It remains to prove that our guess was correct.

For $n = 2$, we have $T(2) \leq c \leq 2c = c \cdot 2 \log_2 2$.

For $n = 4$, we have $T(4) = 2T(2) + 4c \leq 2c + 4c = 6c$.
$4c \log_2 4 = 8c > 6c$.

For $n = 8$, we have $T(8) = 2T(4) + 8c \leq 12c + 8c = 20c$.
$8c \log_2 8 = 24c > 20c$.

Let's use the "guess and verify" technique on this recurrence relation

For simplicity, assume $n = 2^k$ for some integer k.

We guess that: $T(n) \leq cn \log_2 n$ for all $n \geq 2$.

It remains to prove that our guess was correct.

For $n = 2$, we have $T(2) \leq c \leq 2c = c \cdot 2 \log_2 2$.

For $n = 4$, we have $T(4) = 2T(2) + 4c \leq 2c + 4c = 6c$.
$4c \log_2 4 = 8c > 6c$.

For $n = 8$, we have $T(8) = 2T(4) + 8c \leq 12c + 8c = 20c$.
$8c \log_2 8 = 24c > 20c$.

…

Proof by Induction

For simplicity, assume $n = 2^k$ for some integer k.

We guess that: $T(n) \leq cn \log_2 n$ for all $n \geq 2$.

It remains to prove that our guess was correct.

For $n = 2$, we have $T(2) \leq c \leq 2c = cn \log_2 n$.
(Base Case)

Proof by Induction

For simplicity, assume $n = 2^k$ for some integer k.

We guess that: $T(n) \leq cn \log_2 n$ for all $n \geq 2$.

It remains to prove that our guess was correct.

For $n = 2$, we have $T(2) \leq c \leq 2c = cn \log_2 n$.
(Base Case)

Assume that $T(2^{m-1}) \leq c \cdot 2^{m-1} \cdot \log_2 2^{m-1}$ holds for $n = 2^{m-1}$.
(Induction Hypothesis)

Proof by Induction

For simplicity, assume $n = 2^k$ for some integer k.

We guess that: $T(n) \leq cn \log_2 n$ for all $n \geq 2$.

It remains to prove that our guess was correct.

For $n = 2$, we have $T(2) \leq c \leq 2c = cn \log_2 n$.
(Base Case)

Assume that $T(2^{m-1}) \leq c \cdot 2^{m-1} \cdot \log_2 2^{m-1}$ holds for $n = 2^{m-1}$.
(Induction Hypothesis)

We will prove that $T(2^m) \leq c \cdot 2^m \cdot \log_2 2^m$ holds for $n = 2^m$.

Proof by Induction

For $n = 2$, we have $T(2) \leq c \leq 2c = cn \log_2 n$.
(Base Case)

Assume that $T(2^{m-1}) \leq c \cdot 2^{m-1} \cdot \log_2 2^{m-1}$ holds for $n = 2^{m-1}$.
(Induction Hypothesis)

We will prove that $T(2^m) \leq c \cdot 2^m \cdot \log_2 2^m$ holds for $n = 2^{m-1}$.

Proof by Induction

For $n = 2$, we have $T(2) \leq c \leq 2c = cn\log_2 n$.
(Base Case)

Assume that $T(2^{m-1}) \leq c \cdot 2^{m-1} \cdot \log_2 2^{m-1}$ holds for $n = 2^{m-1}$.
(Induction Hypothesis)

We will prove that $T(2^m) \leq c \cdot 2^m \cdot \log_2 2^m$ holds for $n = 2^{m-1}$.

$$\begin{aligned}
T(2^m) &= 2T(2^{m-1}) + c \cdot 2^m \leq 2c \cdot 2^{m-1} \cdot \log_2 2^{m-1} + c \cdot 2^m \\
&= c \cdot 2^m \log_2 2^m - c \cdot 2^m \log_2 2 + c \cdot 2^m = c \cdot 2^m \log_2 m - c \cdot 2^m + c \cdot 2^m \\
&= c \cdot 2^m \log_2 2^m
\end{aligned}$$

Proof by Induction

For $n = 2$, we have $T(2) \leq c \leq 2c = cn\log_2 n$.

(Base Case)

Assume that $T(2^{m-1}) \leq c \cdot 2^{m-1} \cdot \log_2 2^{m-1}$ holds for $n = 2^{m-1}$.

(Induction Hypothesis)

We will prove that $T(2^m) \leq c \cdot 2^m \cdot \log_2 2^m$ holds for $n = 2^{m-1}$.

recurrence

$$
\begin{aligned}
T(2^m) &= 2T(2^{m-1}) + c \cdot 2^m \leq 2c \cdot 2^{m-1} \cdot \log_2 2^{m-1} + c \cdot 2^m \\
&= c \cdot 2^m \log_2 2^m - c \cdot 2^m \log_2 2 + c \cdot 2^m = c \cdot 2^m \log_2 m - c \cdot 2^m + c \cdot 2^m \\
&= c \cdot 2^m \log_2 2^m
\end{aligned}
$$

Proof by Induction

For $n = 2$, we have $T(2) \leq c \leq 2c = cn \log_2 n$.
(Base Case)

Assume that $T(2^{m-1}) \leq c \cdot 2^{m-1} \cdot \log_2 2^{m-1}$ holds for $n = 2^{m-1}$.
(Induction Hypothesis)

We will prove that $T(2^m) \leq c \cdot 2^m \cdot \log_2 2^m$ holds for $n = 2^{m-1}$.

recurrence Ind. Hyp.

$$T(2^m) = 2T(2^{m-1}) + c \cdot 2^m \leq 2c \cdot 2^{m-1} \cdot \log_2 2^{m-1} + c \cdot 2^m$$
$$= c \cdot 2^m \log_2 2^m - c \cdot 2^m \log_2 2 + c \cdot 2^m = c \cdot 2^m \log_2 m - c \cdot 2^m + c \cdot 2^m$$
$$= c \cdot 2^m \log_2 2^m$$

Proof by Induction

For $n = 2$, we have $T(2) \leq c \leq 2c = cn\log_2 n$.
(Base Case)

Assume that $T(2^{m-1}) \leq c \cdot 2^{m-1} \cdot \log_2 2^{m-1}$ holds for
$n = 2^{m-1}$.
(Induction Hypothesis)

We will prove that $T(2^m) \leq c \cdot 2^m \cdot \log_2 2^m$ holds for $n = 2^{m-1}$.

recurrence — Ind. Hyp.

$$T(2^m) = 2T(2^{m-1}) + c \cdot 2^m \leq 2c \cdot 2^{m-1} \cdot \log_2 2^{m-1} + c \cdot 2^m$$
$$= c \cdot 2^m \log_2 2^m - c \cdot 2^m \log_2 2 + c \cdot 2^m = c \cdot 2^m \log_2 m - c \cdot 2^m + c \cdot 2^m$$

logarithm property

$$= c \cdot 2^m \log_2 2^m$$

Let's use the "unrolling" technique on this recurrence relation

Recurrence relation: For some constant c,

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn, & \text{when } n > 2. \\ T(2) \leq c, & \text{otherwise.} \end{cases}$$

Let's consider the simpler version:

$$T(n) = \begin{cases} 2T(n/2) + cn, & \text{when } n > 2. \\ T(2) \leq c, & \text{otherwise.} \end{cases}$$

Solving the Mergesort recursion

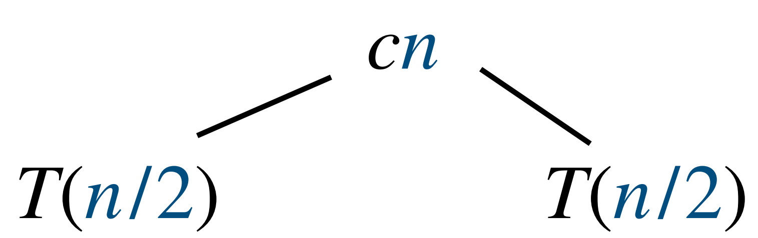
Solving the Mergesort recursion

For simplicity, assume $n = 2^k$ for some integer k.

Solving the Mergesort recursion

For simplicity, assume $n = 2^k$ for some integer k.

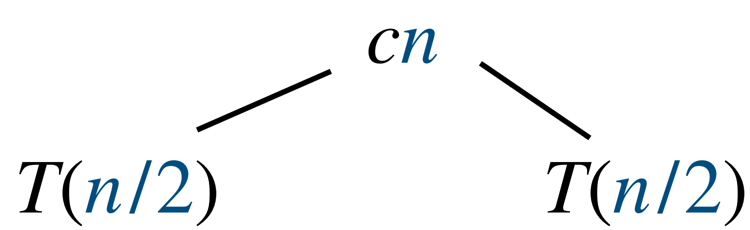
First iteration: Price of cn plus the cost of two subproblems of size $n/2$.



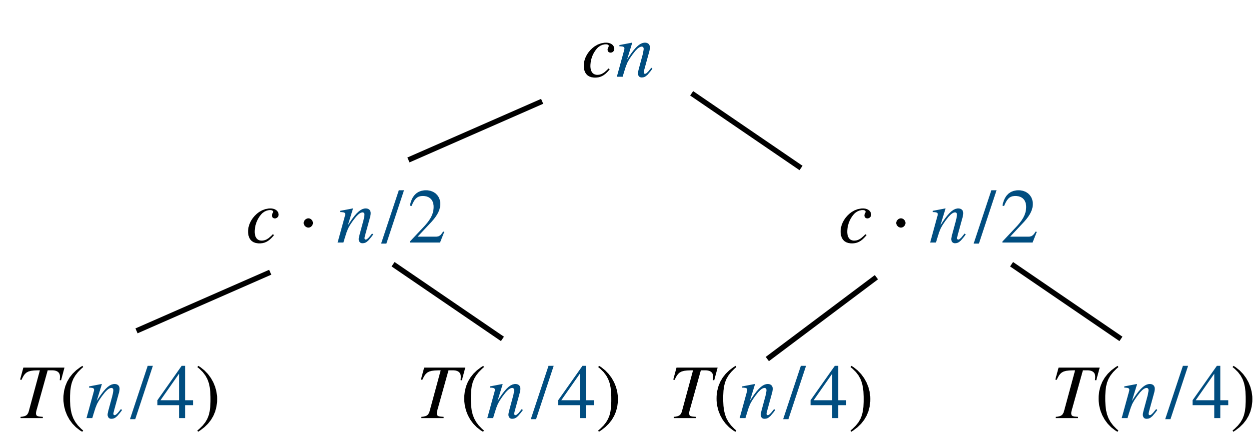
Solving the Mergesort recursion

For simplicity, assume $n = 2^k$ for some integer k.

First iteration: Price of cn plus the cost of two subproblems of size $n/2$.



Second iteration: Price of $c \cdot n/2$ for each subproblem,
plus the cost of two subproblems of size $n/4$



Solving the Mergesort recursion

In total, there will be $\log n + 1$ levels (input halved every time).

Level 0 has cost $C_0(n) = cn$

Level 1 has cost $C_1(n) = 2c \cdot n/2 = cn$

Level 2 has cost $C_2(n) = 4c \cdot n/4 = cn$

Level j has cost $C_j(n) = 2^j c \cdot n/2^j = cn$

The last level has cost cn

Solving the Mergesort recursion

In total, there will be $\log n + 1$ levels (input halved every time).

Level 0 has cost $C_0(n) = cn$

First few levels

Level 1 has cost $C_1(n) = 2c \cdot n/2 = cn$

Level 2 has cost $C_2(n) = 4c \cdot n/4 = cn$

Level j has cost $C_j(n) = 2^j c \cdot n/2^j = cn$

The last level has cost cn

Solving the Mergesort recursion

In total, there will be $\log n + 1$ levels (input halved every time).

Level 0 has cost $C_0(n) = cn$

Level 1 has cost $C_1(n) = 2c \cdot n/2 = cn$

Level 2 has cost $C_2(n) = 4c \cdot n/4 = cn$

First few levels

Level j has cost $C_j(n) = 2^j c \cdot n/2^j = cn$ Identifying a pattern

The last level has cost cn

Solving the **Mergesort** recursion

Solving the Mergesort recursion

Recurrence relation:

$$T(n) = \sum_{j=1}^{\log n+1} C_j(n) + cn = \sum_{j=1}^{\log n+1} cn + cn$$

Solving the Mergesort recursion

Recurrence relation:

Sum the solutions

$$T(n) = \sum_{j=1}^{\log n+1} C_j(n) + cn = \sum_{j=1}^{\log n+1} cn + cn$$

Solving the Mergesort recursion

Recurrence relation:

Sum the solutions

$$T(n) = \sum_{j=1}^{\log n+1} C_j(n) + cn = \sum_{j=1}^{\log n+1} cn + cn$$

The overall running time is $O(n \log n)$.

Guess and verify vs unrolling

Guess and verify vs unrolling

"Unrolling" is not really a formal proof technique.

Guess and verify vs unrolling

"Unrolling" is not really a formal proof technique.

"Identifying a pattern" is informal.

Guess and verify vs unrolling

"Unrolling" is not really a formal proof technique.

"Identifying a pattern" is informal.

It helps us figure out what to "guess".

Guess and verify vs unrolling

"Unrolling" is not really a formal proof technique.

"Identifying a pattern" is informal.

It helps us figure out what to "guess".

Then we can "verify".

Guess and verify vs unrolling

“Unrolling” is not really a formal proof technique.

“Identifying a pattern” is informal.

It helps us figure out what to “guess”.

Then we can “verify”.

For a formal proof, those techniques could be used together.

The Quicksort algorithm

Algorithm **Quicksort**(**A**[$i,\ldots,j$])

y = **Partition**(**A**[$i,\ldots,j$])

Quicksort(**A**[$i,\ldots,y\text{-}1$])

Quicksort(**A**[$y+1,\ldots,j$])

The Quicksort algorithm

Algorithm **Quicksort**(**A**[*i*,...,*j*])

y = **Partition**(**A**[*i*,...,*j*])

Quicksort(**A**[*i*,...,*y*-1])

Quicksort(**A**[*y*+1,...,*j*])

$$T(n) \leq T(n_1) + T(n_2) + cn$$

Running time of Quicksort

Running time of Quicksort

Mergesort: $T(n) \leq 2T(n) + cn$

Running time of Quicksort

Mergesort: $T(n) \leq 2T(n) + cn$

Quicksort: $T(n) \leq T(n_1) + T(n_2) + cn$

Running time of Quicksort

Mergesort: $T(n) \leq 2T(n) + cn$

Quicksort: $T(n) \leq T(n_1) + T(n_2) + cn$

When $n_1 = n_2$, the running time is the same as **Mergesort.**

Running time of Quicksort

Mergesort: $T(n) \leq 2T(n) + cn$

Quicksort: $T(n) \leq T(n_1) + T(n_2) + cn$

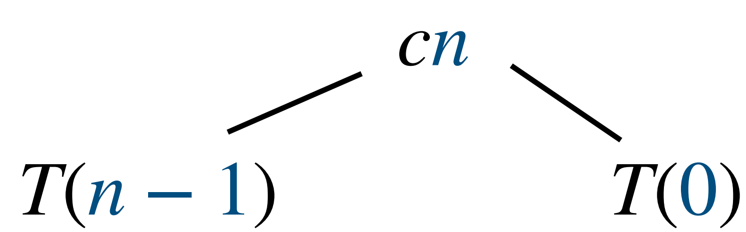
When $n_1 = n_2$, the running time is the same as **Mergesort.**

What is the worst possible running time?

“Unrolling”

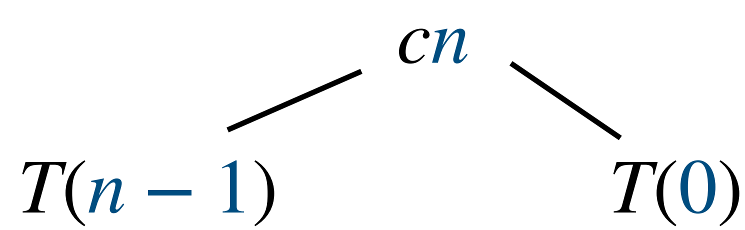
"Unrolling"

First iteration: Price of cn plus the cost of two subproblems of size $n - 1$ and 0.

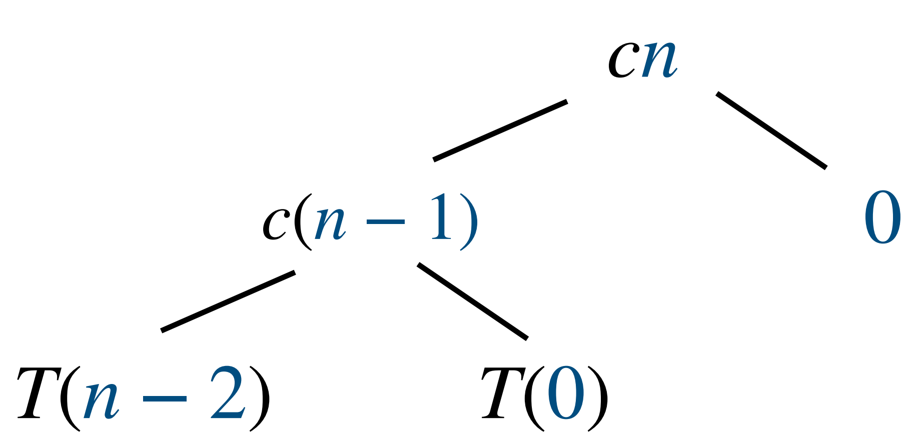


“Unrolling”

First iteration: Price of cn plus the cost of two subproblems of size $n - 1$ and 0.



Second iteration: Price of $c(n-1)$, plus the cost of two subproblems of size $n - 2$ and 0.

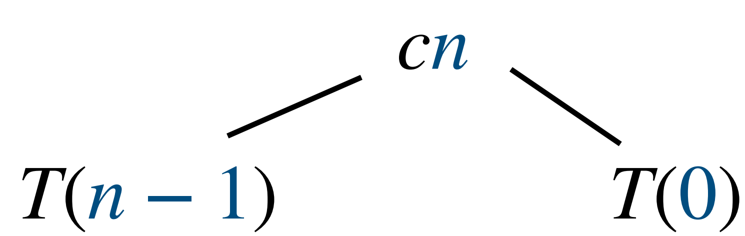


Solving the Quicksort recursion

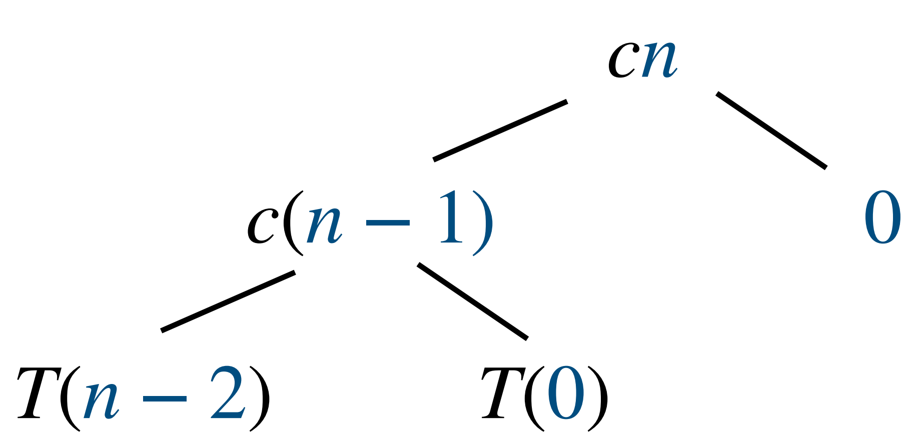
How many levels do we have in total?

“Unrolling”

First iteration: Price of cn plus the cost of two subproblems of size $n-1$ and 0.



Second iteration: Price of $c(n-1)$, plus the cost of two subproblems of size $n-2$ and 0.



Solving the Quicksort recursion

How many levels do we have in total? n levels.

Level 0 has cost $C_0(n) = cn$

Level 1 has cost $C_1(n) = c(n-1)$

Level 2 has cost $C_2(n) = c(n-2)$

Level j has cost $C_j(n) = c(n-j)$

The last level has cost c

Solving the Quicksort recursion

Solving the Quicksort recursion

Recurrence relation:

$$T(n) = \sum_{j=1}^{n-1} C_j(n) = c(n + n - 1 + n - 2 + \ldots + 1)$$

$$= c\frac{n(n+1)}{2} \leq cn^2$$

Solving the Quicksort recursion

Recurrence relation:

$$T(n) = \sum_{j=1}^{n-1} C_j(n) = c(n + n - 1 + n - 2 + \ldots + 1)$$

$$= c\frac{n(n+1)}{2} \leq cn^2$$

The overall running time is $O(n^2)$.

Lower bound for `Quicksort`

Lower bound for `Quicksort`

Is our analysis *tight* enough? Could it be that there is a better analysis that shows a $o(n^2)$ running time?

Lower bound for Quicksort

Is our analysis *tight* enough? Could it be that there is a better analysis that shows a $o(n^2)$ running time?

Can we show a lower bound of $\Omega(n^2)$ on the running time of Quicksort?

Upper and Lower (Worst-Case) Bounds

Upper Bound $O(g_1(n))$: On *any possible input* to the problem, our algorithm will take time (at most) $O(g_1(n))$.

Lower Bound $\Omega(g_2(n))$: There *exists at least one input* to the problem, on which our algorithm will take time (at least) $\Omega(g_2(n))$.

When $g_1(n) = g_2(n)$, we say that our running time analysis is *tight*, and we have fully understood the (asymptotic, worst-case) running time of the algorithm.

Lower bound for **Quicksort**

Is your analysis *tight* enough? Could it be that there is a better analysis that shows a $o(n^2)$ running time?

Can we show a lower bound of $\Omega(n^2)$ on the running time of **Quicksort**?

Lower bound for **Quicksort**

Is your analysis *tight* enough? Could it be that there is a better analysis that shows a $o(n^2)$ running time?

Can we show a lower bound of $\Omega(n^2)$ on the running time of **Quicksort**?

Q: Can you think of an input where **Quicksort** takes time $\Omega(n^2)$?

Upper and Lower (Worst-Case) Bounds

Upper Bound $O(g_1(n))$: On *any possible input* to the problem, our algorithm will take time (at most) $O(g_1(n))$.

Lower Bound $\Omega(g_2(n))$: There *exists at least one input* to the problem, on which our algorithm will take time (at least) $\Omega(g_2(n))$.

When $g_1(n) = g_2(n)$, we say that our running time analysis is *tight*, and we have fully understood the (asymptotic, worst-case) running time of the algorithm.

Upper and Lower (Worst-Case) Bounds *for algorithms*

Upper Bound $O(g_1(n))$: On *any possible input* to the problem, *our algorithm* will take time (at most) $O(g_1(n))$.

Lower Bound $\Omega(g_2(n))$: There *exists at least one input* to the problem, on which *our algorithm* will take time (at least) $\Omega(g_2(n))$.

When $g_1(n) = g_2(n)$, we say that our running time analysis is *tight*, and we have fully understood the (asymptotic, worst-case) running time of *the algorithm*.

Upper and Lower (Worst-Case) Bounds *for problems*

Upper and Lower (Worst-Case) Bounds *for problems*

Upper Bound $O(g_1(n))$: *There exists an algorithm* which, on *any possible input* to the problem, takes time (at most) $O(g_1(n))$.

Upper and Lower (Worst-Case) Bounds *for problems*

Upper Bound $O(g_1(n))$: *There exists an algorithm* which, on *any possible input* to the problem, takes time (at most) $O(g_1(n))$.

Lower Bound $\Omega(g_2(n))$: *For any algorithm*, there *exists at least one input* to the problem, on which the algorithm will take time (at least) $\Omega(g_2(n))$.

Upper and Lower (Worst-Case) Bounds *for problems*

Upper Bound $O(g_1(n))$: *There exists an algorithm* which, on *any possible input* to the problem, takes time (at most) $O(g_1(n))$.

Lower Bound $\Omega(g_2(n))$: *For any algorithm*, there *exists at least one input* to the problem, on which the algorithm will take time (at least) $\Omega(g_2(n))$.

When $g_1(n) = g_2(n)$, we say that our running time analysis is *tight*, and we have fully understood the (asymptotic, worst-case) running time of *the problem*.

For us here concretely

For us here concretely

Upper Bound $O(n \log n)$: We have identified an algorithm **(Mergesort)** that has worst-case running time $O(n \log n)$.

For us here concretely

Upper Bound $O(n \log n)$: We have identified an algorithm (**Mergesort**) that has worst-case running time $O(n \log n)$.

Lower Bound $\Omega(n \log n)$: We need to prove that *every algorithm* has worst-case running time $\Omega(n \log n)$.

For us here concretely

Upper Bound $O(n \log n)$: We have identified an algorithm (**Mergesort**) that has worst-case running time $O(n \log n)$.

Lower Bound $\Omega(n \log n)$: We need to prove that *every algorithm* has worst-case running time $\Omega(n \log n)$.

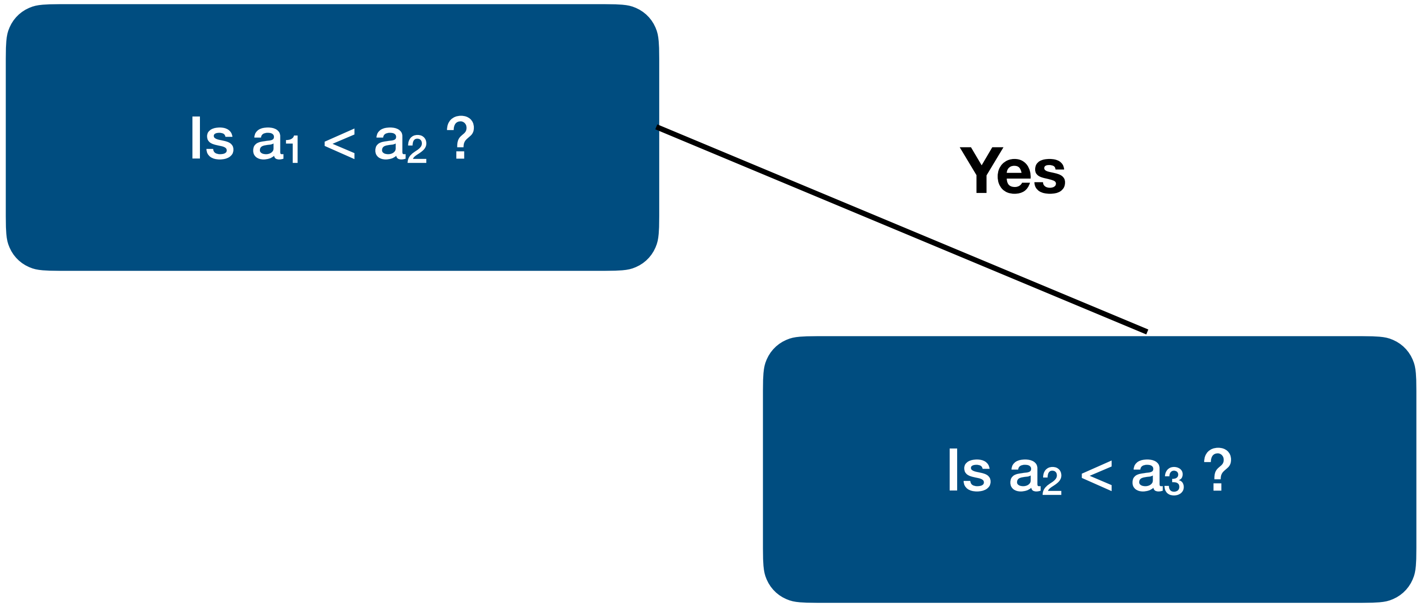
In other words, we will prove that there is no algorithm that is *asymptotically better* than **Mergesort**.

Lower bound for sorting

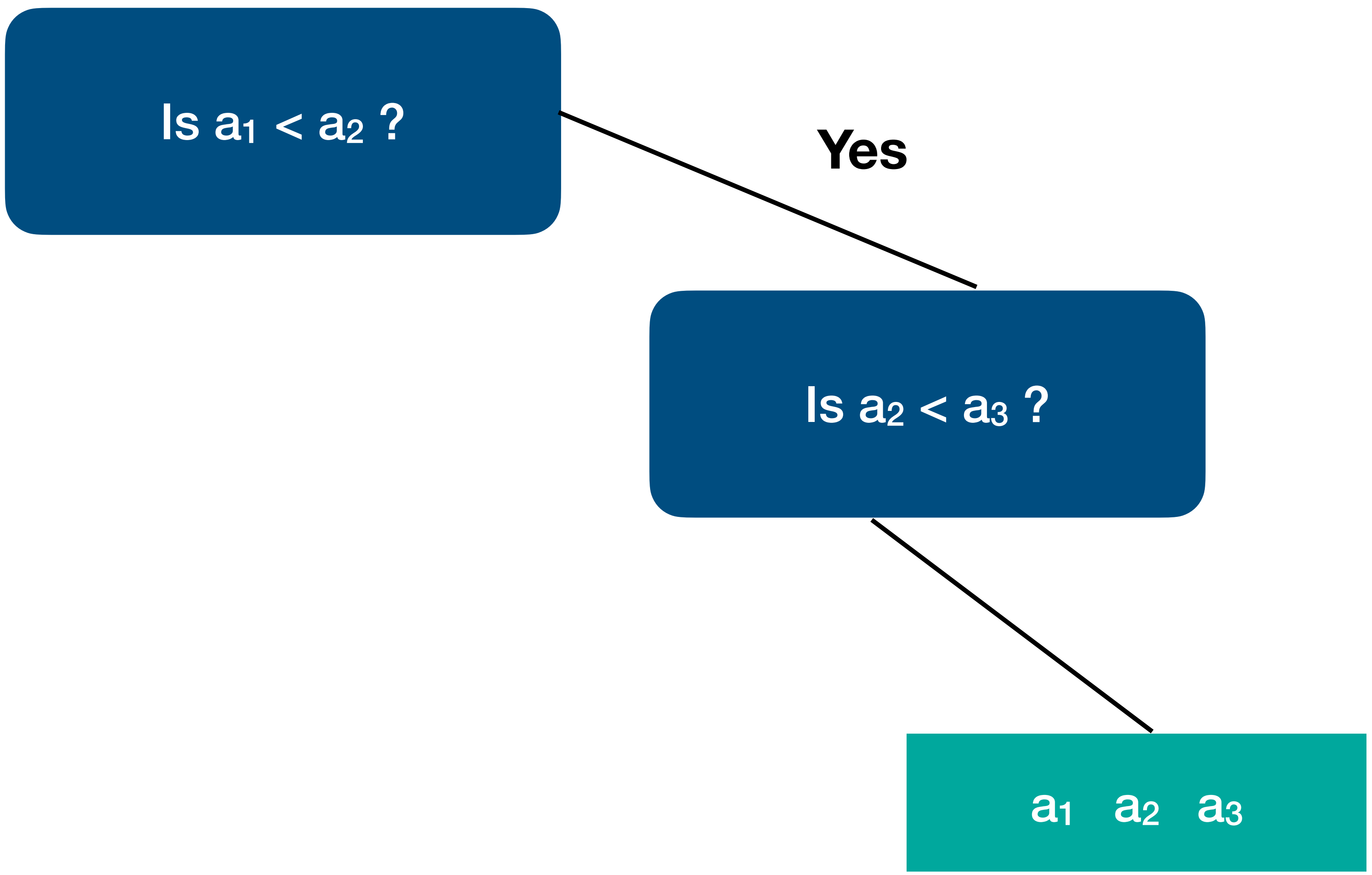
Lower bound for sorting

Is $a_1 < a_2$?

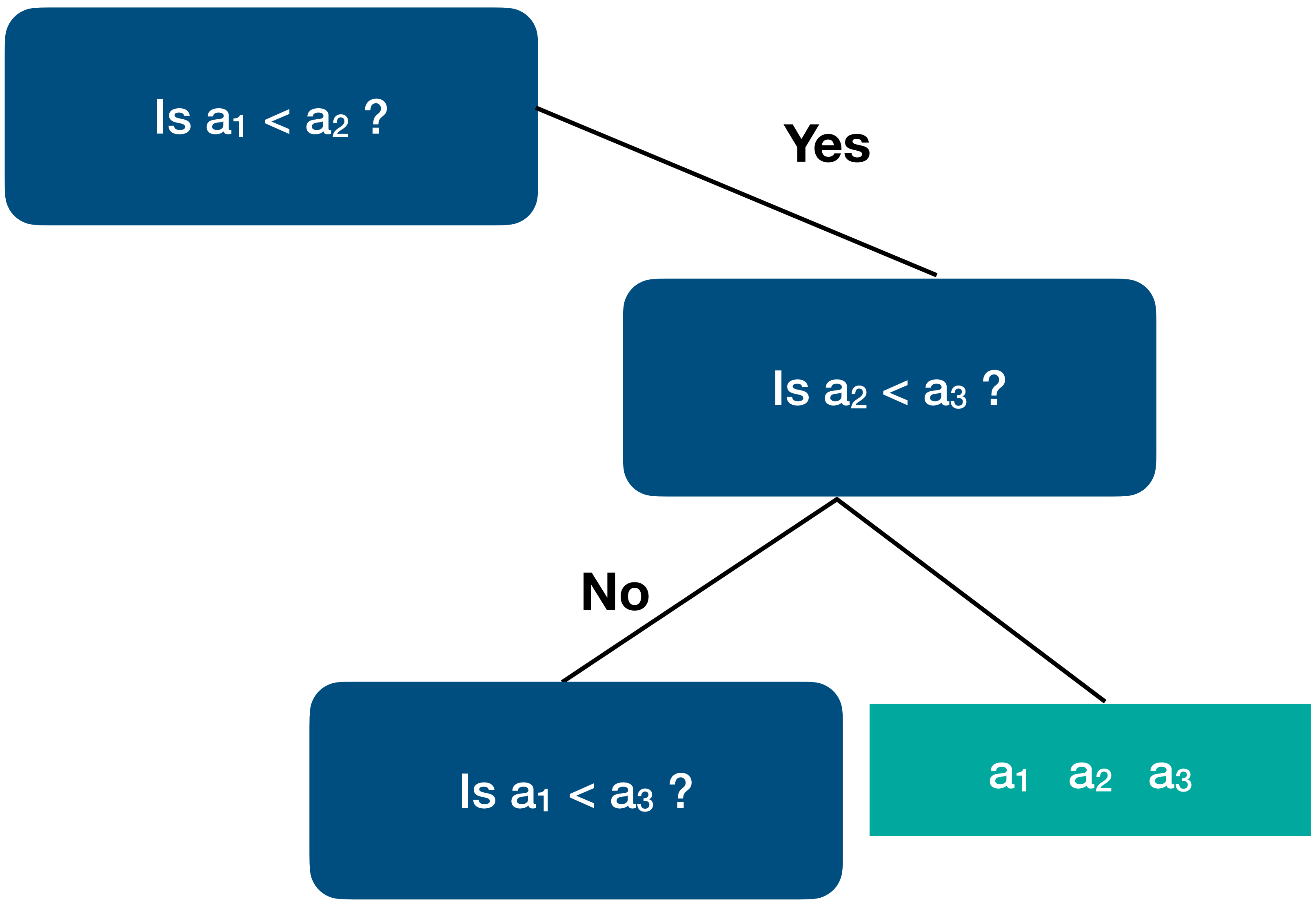
Lower bound for sorting



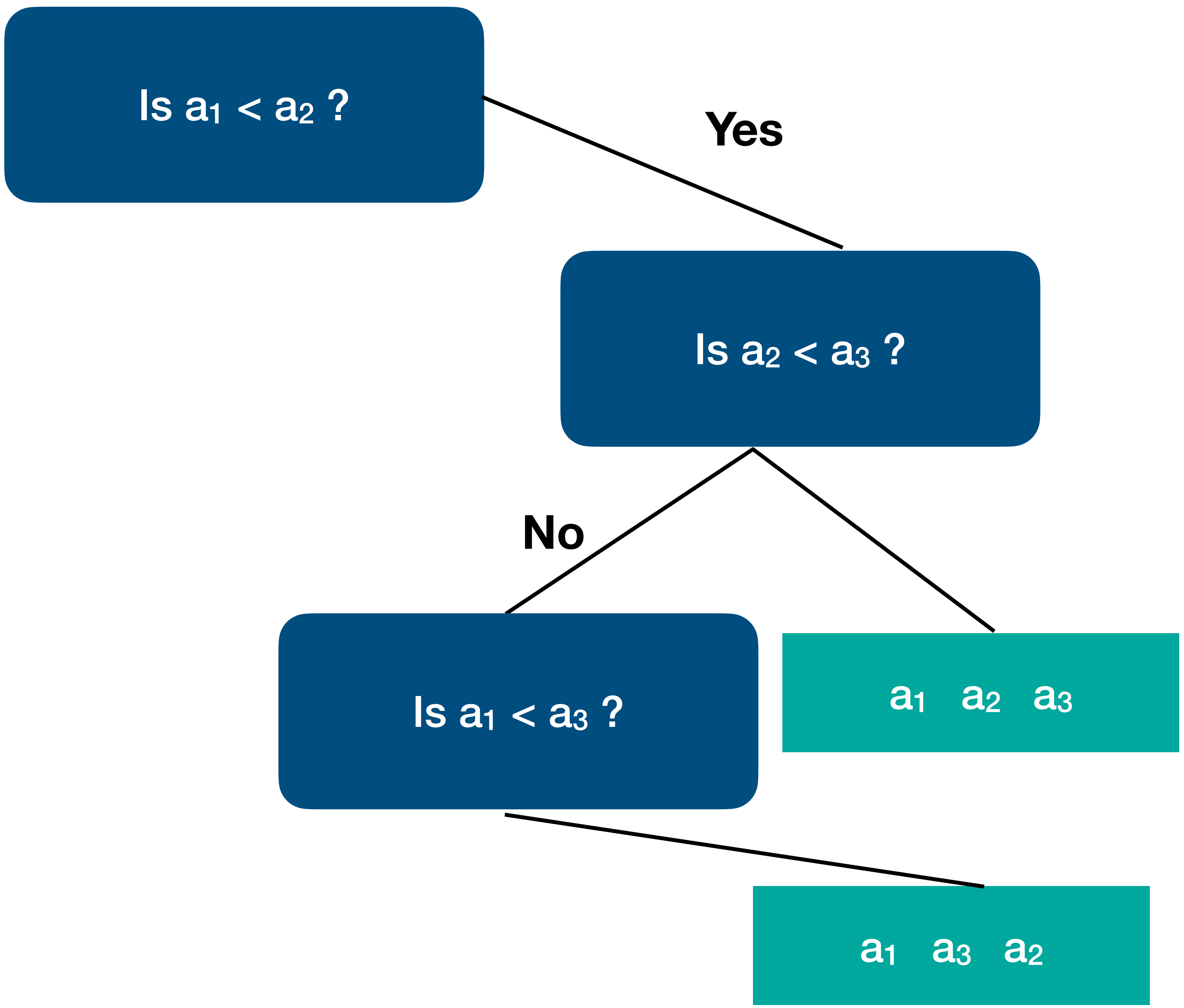
Lower bound for sorting



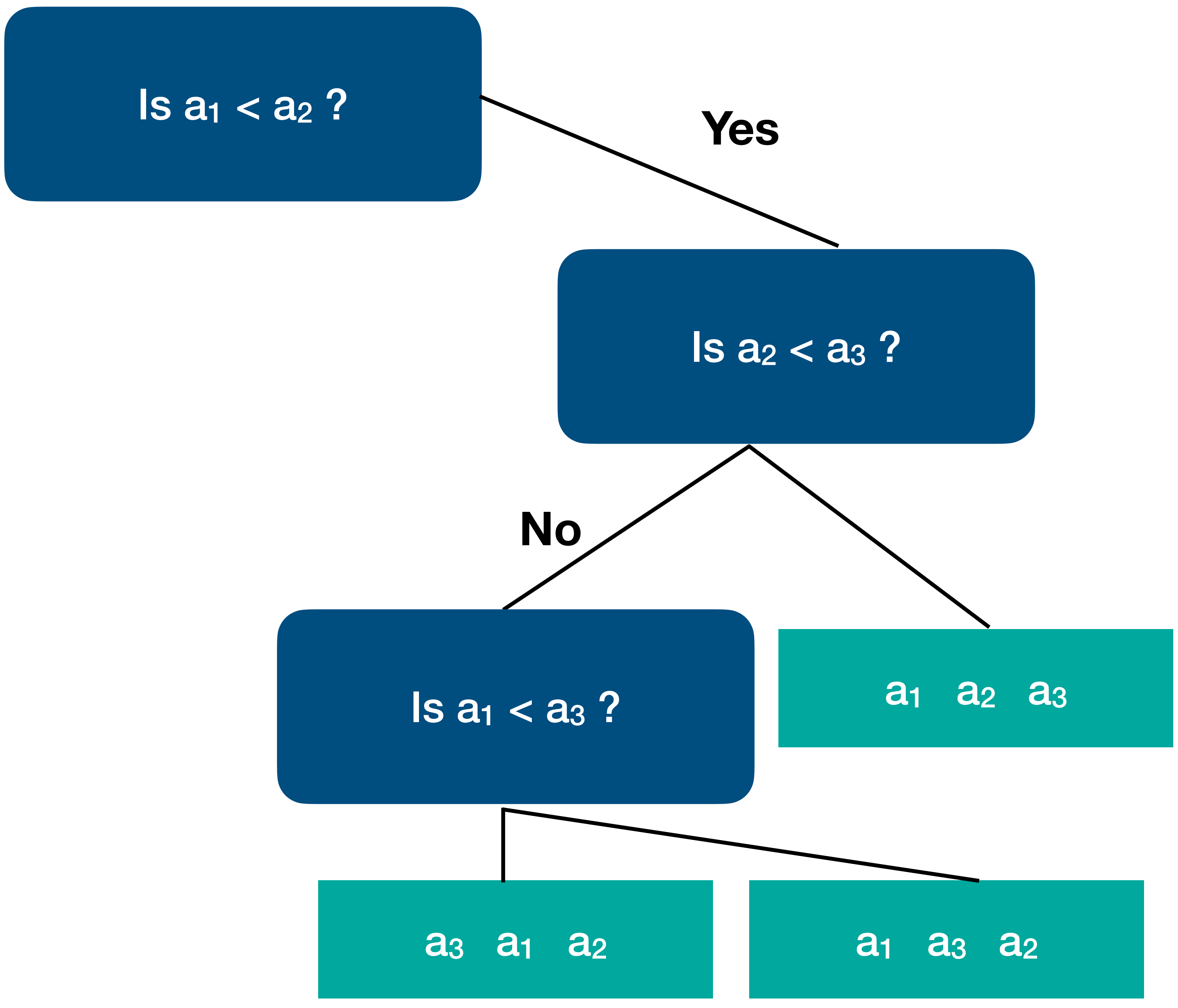
Lower bound for sorting



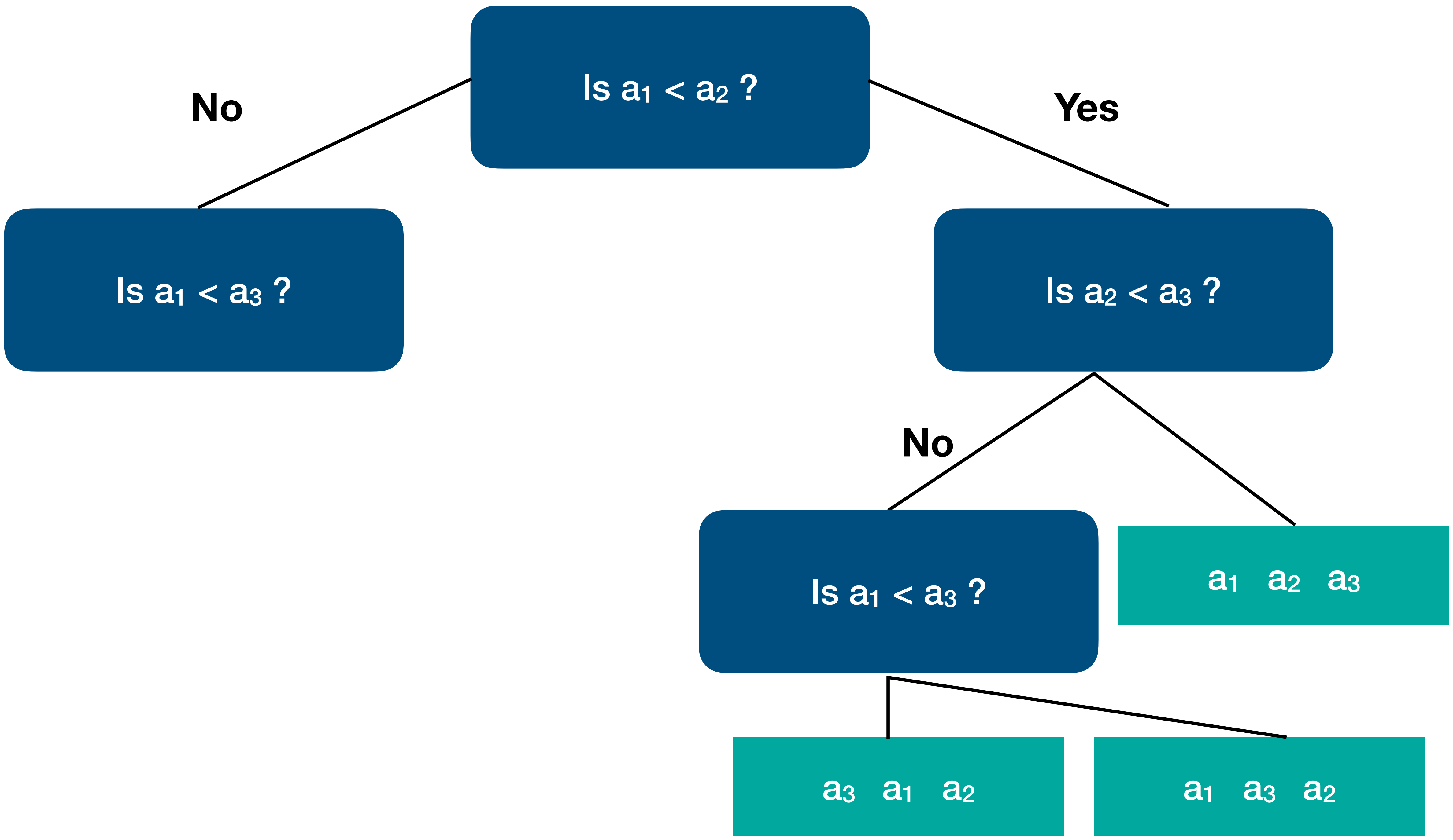
Lower bound for sorting



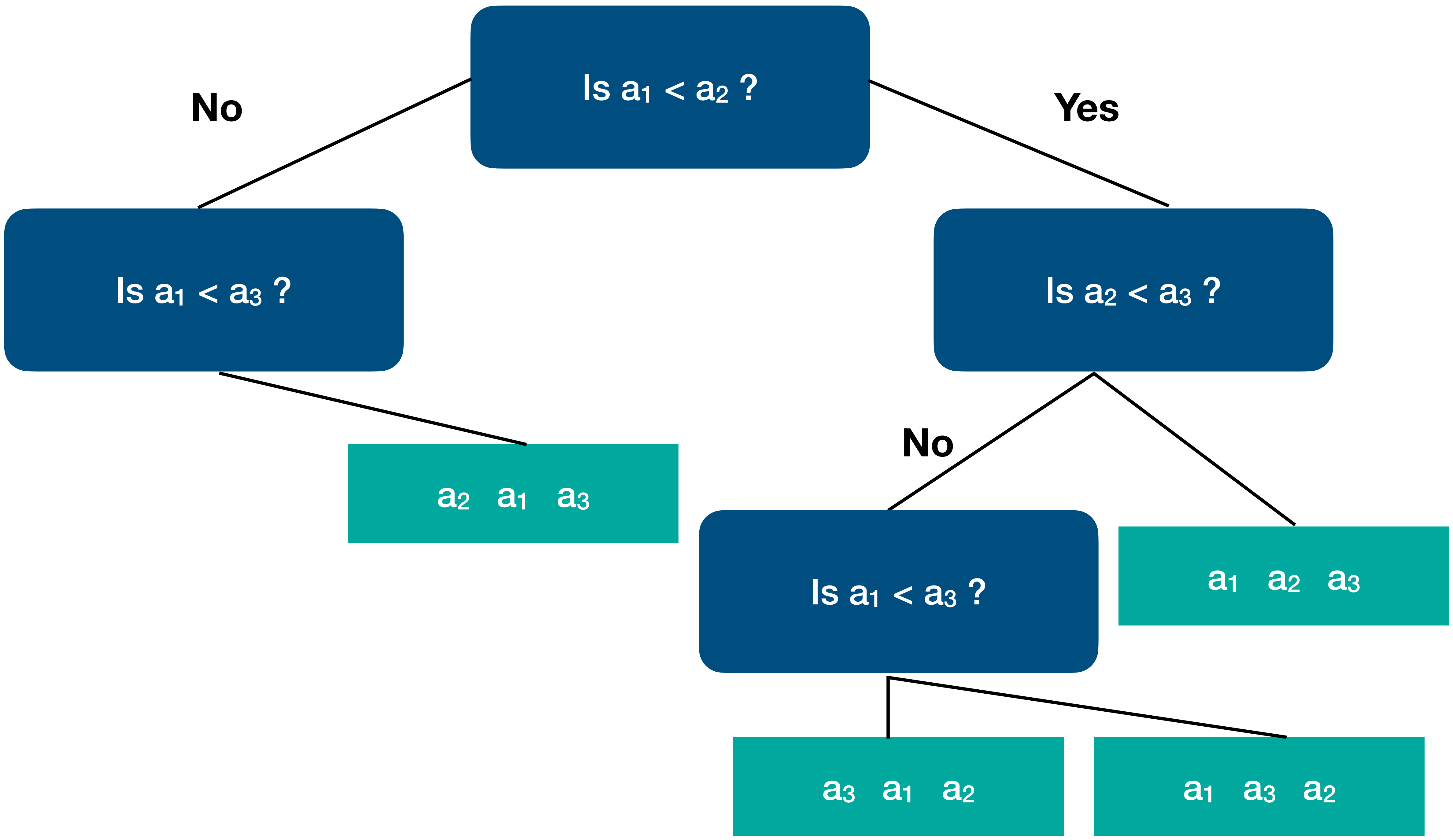
Lower bound for sorting



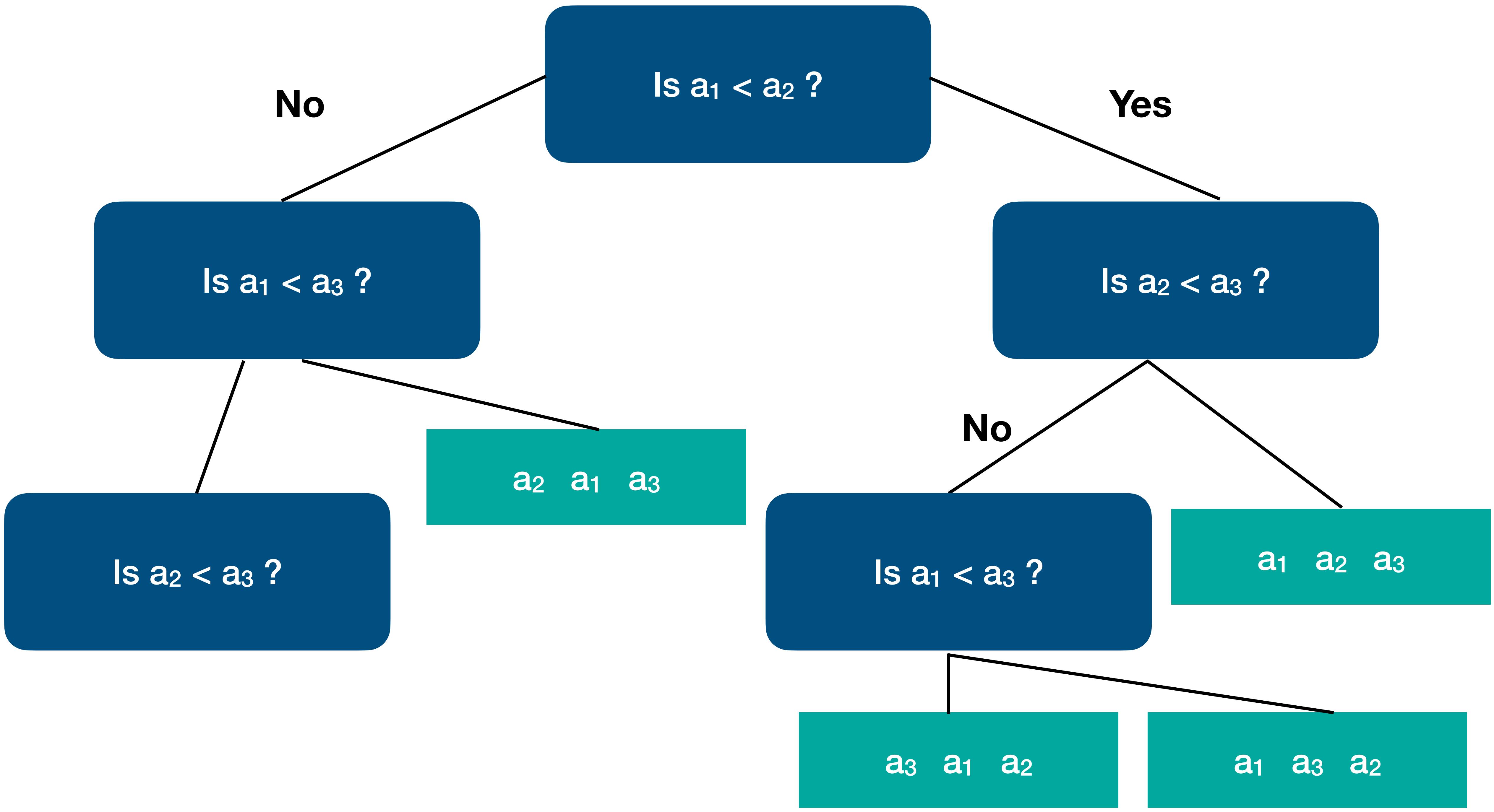
Lower bound for sorting



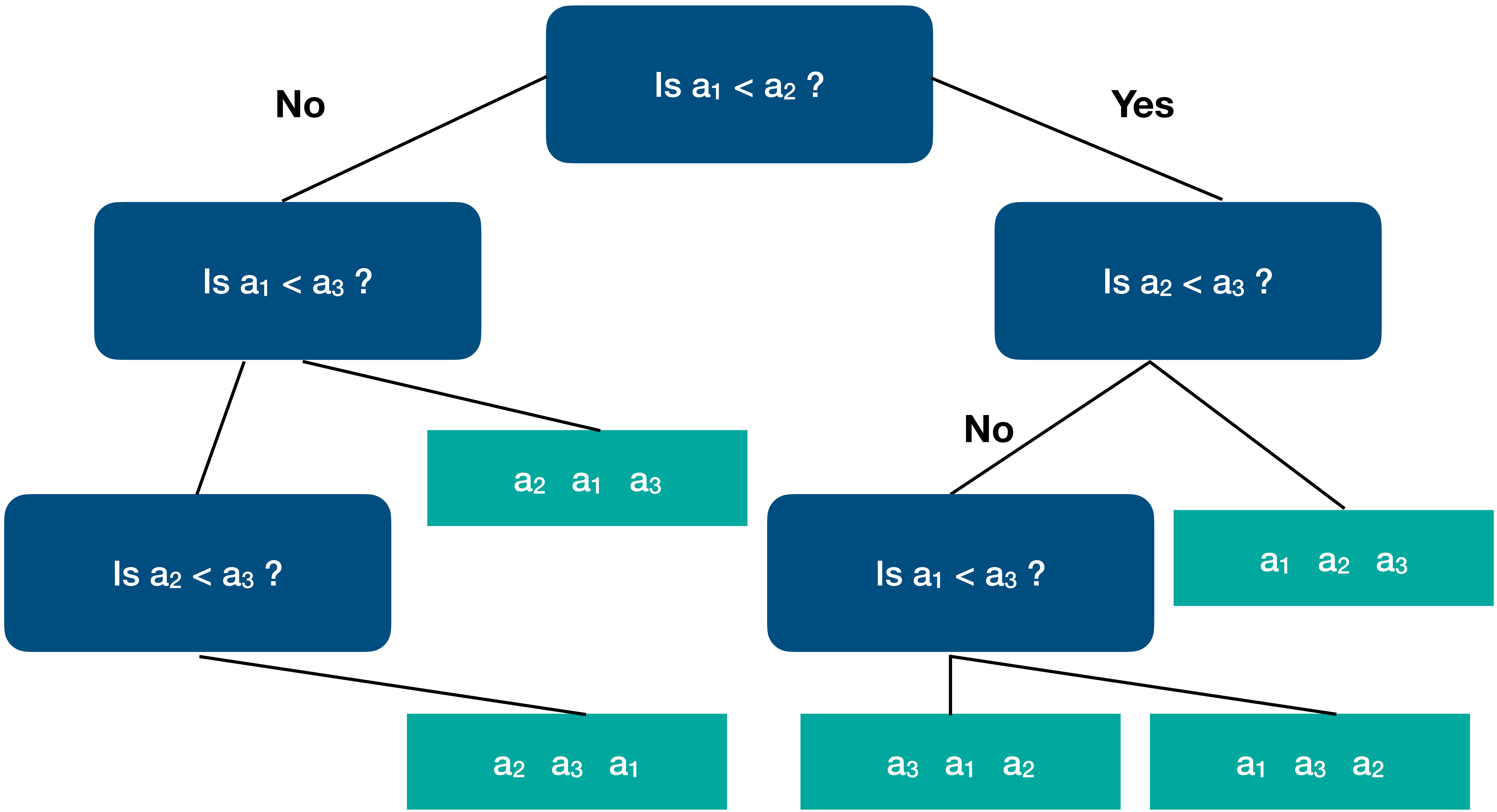
Lower bound for sorting



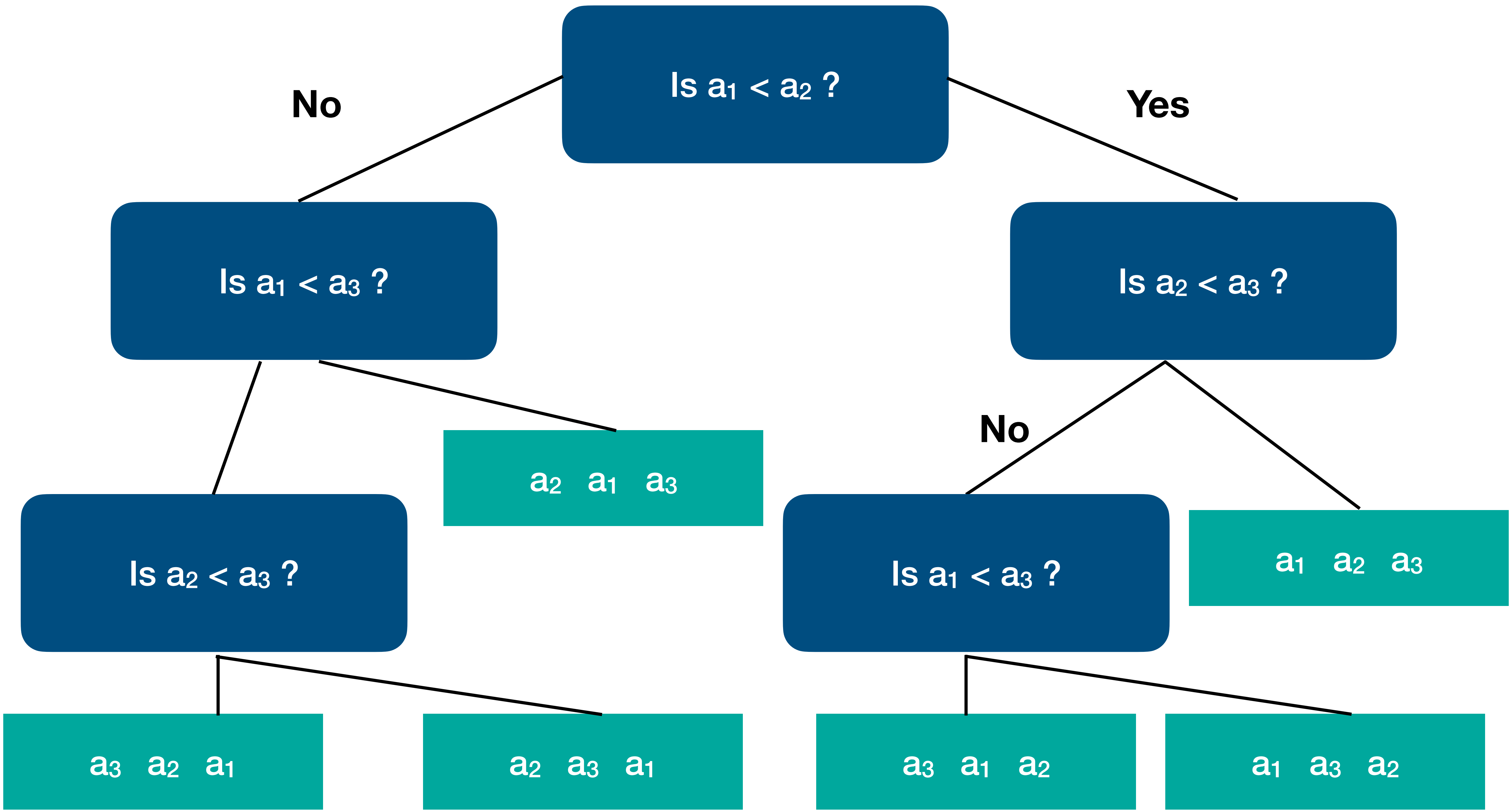
Lower bound for sorting



Lower bound for sorting



Lower bound for sorting



Lower bound for sorting

Lower bound for sorting

We need as many comparisons as the *depth* of the tree (length of the longest path from the root to the leaves).

Lower bound for sorting

We need as many comparisons as the *depth* of the tree (length of the longest path from the root to the leaves).

The decision tree has $n!$ leaves

Lower bound for sorting

We need as many comparisons as the *depth* of the tree (length of the longest path from the root to the leaves).

The decision tree has $n!$ leaves

A leaf is a permutation of $\{a_1, a_2, \ldots, a_n\}$

Lower bound for sorting

We need as many comparisons as the *depth* of the tree (length of the longest path from the root to the leaves).

The decision tree has $n!$ leaves

A leaf is a permutation of $\{a_1, a_2, \ldots, a_n\}$

Every possible permutation can appear as a leaf, since every possible permutation is a valid output.

Lower bound for sorting

Lower bound for sorting

Fact: Every binary tree of depth d has at most 2^d leaves.

Lower bound for sorting

Think about it at home! Try to prove it using induction.

Fact: Every binary tree of depth d has at most 2^d leaves.

Lower bound for sorting

Think about it at home! Try to prove it using induction.

Fact: Every binary tree of depth d has at most 2^d leaves.

Therefore the minimum number of comparisons is $\log_2(n!)$

Lower bound for sorting

Think about it at home! Try to prove it using induction.

Fact: Every binary tree of depth d has at most 2^d leaves.

Therefore the minimum number of comparisons is $\log_2(n!)$

We claim that $\log_2(n!) = \Omega(n \log n)$

Lower bound for sorting

Think about it at home! Try to prove it using induction.

Fact: Every binary tree of depth d has at most 2^d leaves.

Therefore the minimum number of comparisons is $\log_2(n!)$

We claim that $\log_2(n!) = \Omega(n \log n)$

$$\log_2(n!) = \log_2 (1 \cdot 2 \cdot , \ldots, \cdot n)$$

Lower bound for sorting

Think about it at home! Try to prove it using induction.

Fact: Every binary tree of depth d has at most 2^d leaves.

Therefore the minimum number of comparisons is $\log_2(n!)$

We claim that $\log_2(n!) = \Omega(n \log n)$

$$\log_2(n!) = \log_2\left(1 \cdot 2 \cdot , \ldots, \cdot n\right)$$
$$= \log_2(1) + \log_2(2) + \ldots + \log_2(n)$$

Lower bound for sorting

Think about it at home! Try to prove it using induction.

Fact: Every binary tree of depth d has at most 2^d leaves.

Therefore the minimum number of comparisons is $\log_2(n!)$

We claim that $\log_2(n!) = \Omega(n \log n)$

$$\begin{aligned}\log_2(n!) &= \log_2 (1 \cdot 2 \cdot, \ldots, \cdot n) \\ &= \log_2(1) + \log_2(2) + \ldots + \log_2(n) \\ &\geq \log_2(n/2) + \ldots + \log_2(n) \text{ *(half)*}\end{aligned}$$

Lower bound for sorting

Think about it at home! Try to prove it using induction.

Fact: Every binary tree of depth d has at most 2^d leaves.

Therefore the minimum number of comparisons is $\log_2(n!)$

We claim that $\log_2(n!) = \Omega(n \log n)$

$$\log_2(n!) = \log_2 (1 \cdot 2 \cdot , \ldots, \cdot n)$$
$$= \log_2(1) + \log_2(2) + \ldots + \log_2(n)$$
$$\geq \log_2(n/2) + \ldots + \log_2(n) \quad \textit{(half)}$$
$$\geq \log_2(n/2) + \ldots + \log_2(n/2) = (n/2) \log_2(n/2)$$

Quick Note

Quick Note

This lower bound is often referred to as a lower bound for "*comparison based sorting*".

Quick Note

This lower bound is often referred to as a lower bound for "*comparison based sorting*".

But it is indeed a general sorting lower bound, as in the general case, the order between elements is defined via comparisons between them.

Quick Note

This lower bound is often referred to as a lower bound for "*comparison based sorting*".

But it is indeed a general sorting lower bound, as in the general case, the order between elements is defined via comparisons between them.

If our array has some specific properties, then we can sort in $O(n)$ time, using algorithms that do not apply to the general problem.

Quick Note

This lower bound is often referred to as a lower bound for "*comparison based sorting*".

But it is indeed a general sorting lower bound, as in the general case, the order between elements is defined via comparisons between them.

If our array has some specific properties, then we can sort in $O(n)$ time, using algorithms that do not apply to the general problem.

e.g., **CountingSort**

Proving lower bounds

Proving lower bounds

Consider some criterion A that we would like to minimise (could be running time, memory, etc).

Proving lower bounds

Consider some criterion A that we would like to minimise (could be running time, memory, etc).

We want to find the best algorithm (asymptotically) for criterion A.

Proving lower bounds

Consider some criterion A that we would like to minimise (could be running time, memory, etc).

We want to find the best algorithm (asymptotically) for criterion A.

The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.

Proving lower bounds

Consider some criterion A that we would like to minimise (could be running time, memory, etc).

We want to find the best algorithm (asymptotically) for criterion A.

The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.

Upper bound: We construct an algorithm that has performance $O(g(n))$ for criterion A.

Proving lower bounds

Consider some criterion A that we would like to minimise (could be running time, memory, etc).

We want to find the best algorithm (asymptotically) for criterion A.

The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.

Upper bound: We construct an algorithm that has performance $O(g(n))$ for criterion A.

Lower bound: We show that for any algorithm, the performance for criterion A is $\Omega(g(n))$.

Proving lower bounds

The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.

Proving lower bounds

The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.

How do we find this function?

Proving lower bounds

The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.

How do we find this function?

No easy answer!

Proving lower bounds

The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.

How do we find this function?

No easy answer!

We try to design algorithms which are as good as possible and when we feel that we can not improve more, we try to prove the *matching* lower bound.

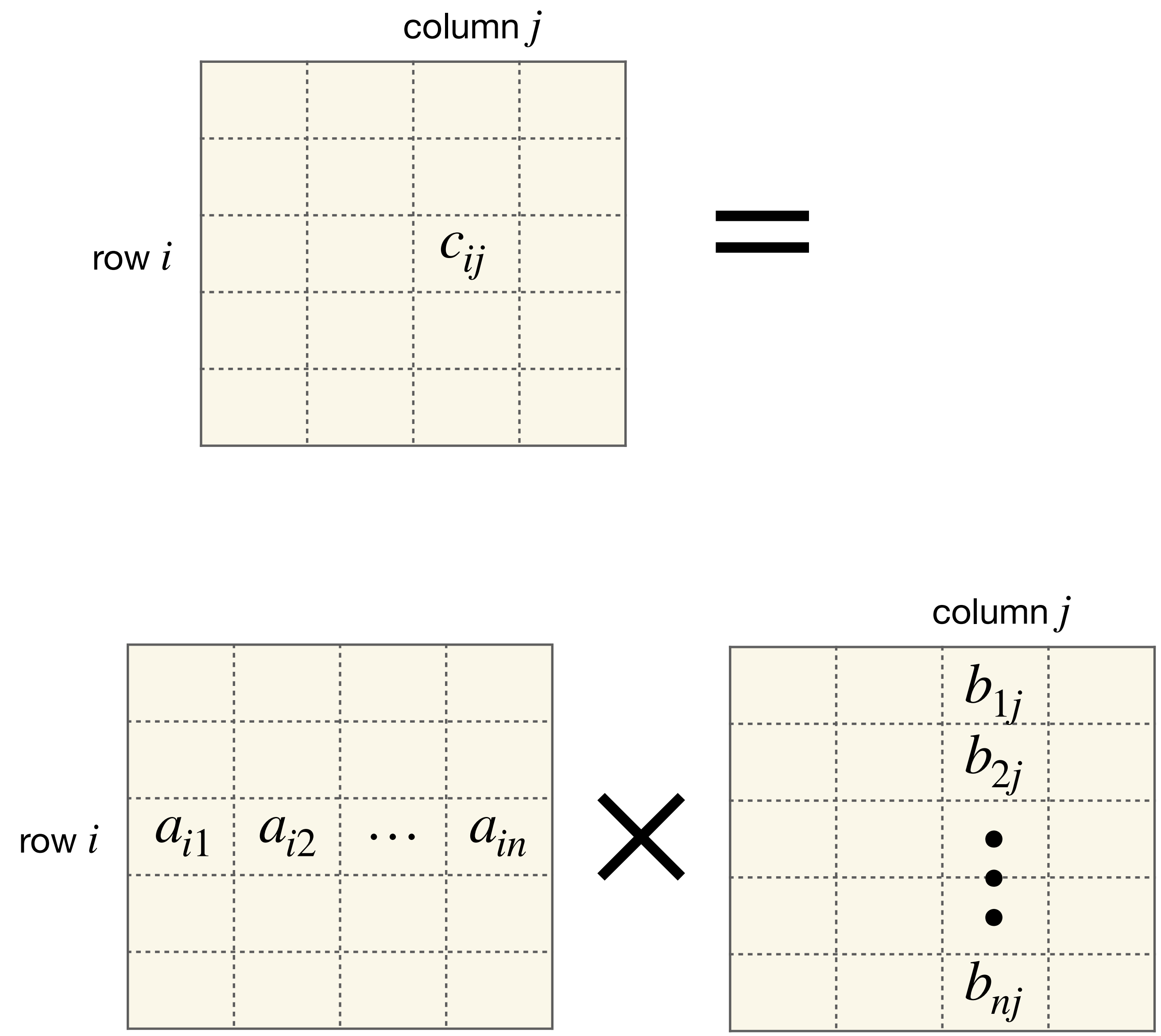
Matrix Multiplication

Matrix Multiplication

Assume that we have two square $(n \times n)$-matrices $A = (a_{ij})_{1 \le i,j \le n}$ and $B = (b_{ij})_{1 \le i,j \le n}$

The product of A and B is the $(n \times n)$-matrix $C = (c_{ij})_{1 \le i,j \le n}$ with entries

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$



A straightforward approach

Compute the sum of pairwise products $a_{ik} \cdot b_{kj}$ for each entry c_{ij} of C.

We have n^2 entries, and n pairwise products for each entry.

A straightforward approach

Compute the sum of pairwise products $a_{ik} \cdot b_{kj}$ for each entry c_{ij} of C.

We have n^2 entries, and n pairwise products for each entry.

Matrix-Multiply (A, B)

for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
        $c_{ij} = 0$
        for $k = 1$ **to** n **do**
            $c_{ij} = c_{ij} + a_{ik} + b_{kj}$

return $C = (c_{ij})_{1 \leq i,j \leq n}$

A straightforward approach

Compute the sum of pairwise products $a_{ik} \cdot b_{kj}$ for each entry c_{ij} of C.

We have n^2 entries, and n pairwise products for each entry.

Matrix-Multiply (A, B)

for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
        $c_{ij} = 0$
        for $k = 1$ **to** n **do**
            $c_{ij} = c_{ij} + a_{ik} + b_{kj}$

return $C = (c_{ij})_{1 \leq i,j \leq n}$

Running time: $\Theta(n^3)$

A naive D&C approach

Suppose we divide our matrices A and B as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

We can write C as:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}$$

A naive D&C approach

Suppose we divide our matrices A and B as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

We will assume from now on that $n = 2^k$ for some k.

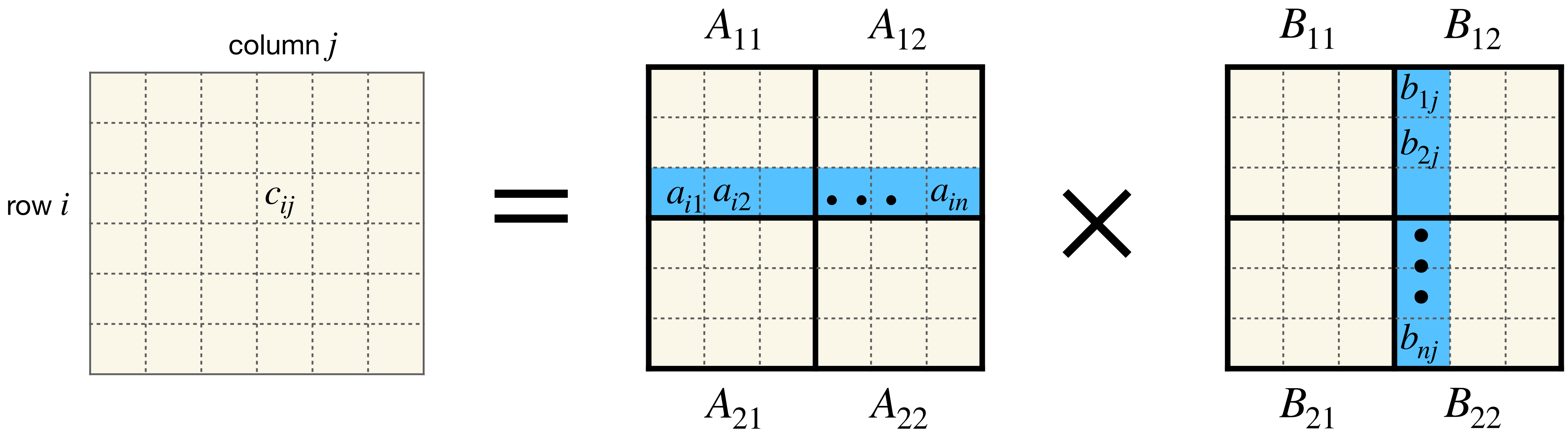
We can write C as:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}$$

Pictorially

Suppose we divide our matrices A and B as follows:

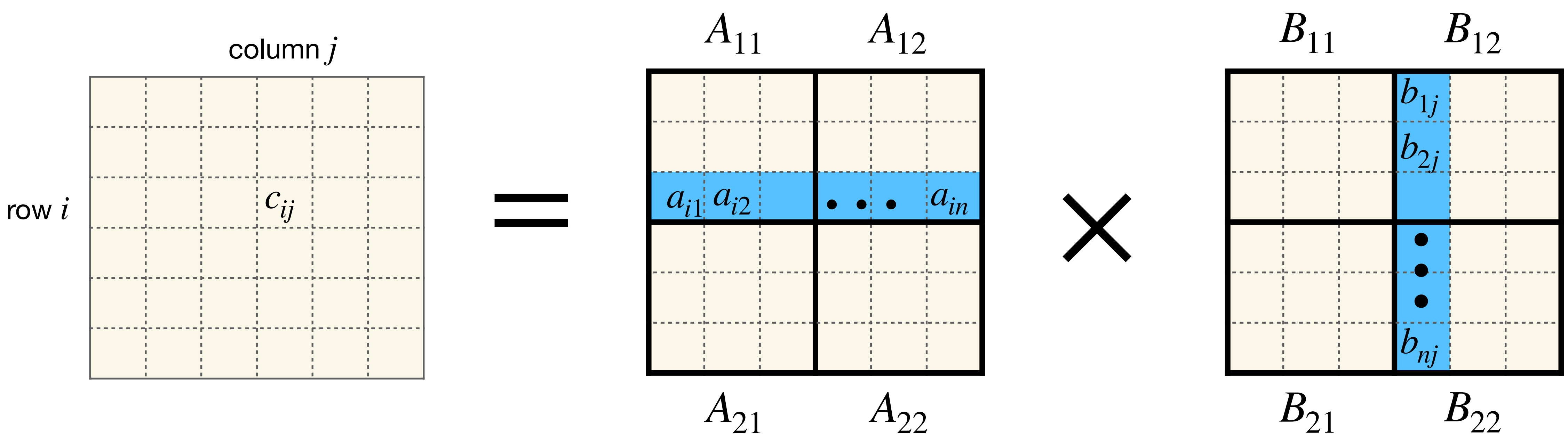


Suppose $i \leq n/2$ and $j > n/2$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = \sum_{k=1}^{n/2} a_{ik} b_{kj} + \sum_{n/2+1}^{n} a_{ik} b_{kj}$$

Pictorially

Suppose we divide our matrices A and B as follows:

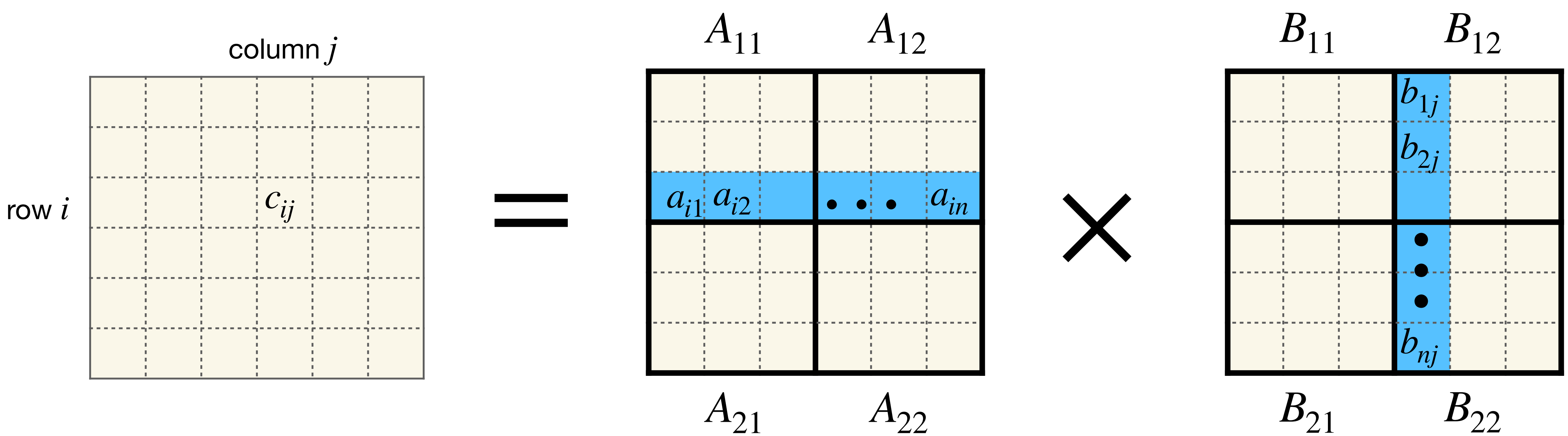


Suppose $i \leq n/2$ and $j > n/2$

$$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj} = \underbrace{\sum_{k=1}^{n/2} a_{ik}b_{kj}}_{\in A_{11} \cdot B_{12}} + \sum_{n/2+1}^{n} a_{ik}b_{kj}$$

Pictorially

Suppose we divide our matrices A and B as follows:



Suppose $i \le n/2$ and $j > n/2$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = \underbrace{\sum_{k=1}^{n/2} a_{ik} b_{kj}}_{\in A_{11} \cdot B_{12}} + \underbrace{\sum_{n/2+1}^{n} a_{ik} b_{kj}}_{\in A_{12} \cdot B_{22}}$$

The D&C algorithm

Matrix-Multiply-DC (A, B)

if $n = 1$, **do**

$c_{11} = a_{11} \cdot b_{11}$

return c_{11}

Partition $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ **and** $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

$C_{11} =$ **Matrix-Multiply-DC** (A_{11}, B_{11}) + **Matrix-Multiply-DC** (A_{12}, B_{21})
$C_{12} =$ **Matrix-Multiply-DC** (A_{11}, B_{12}) + **Matrix-Multiply-DC** (A_{12}, B_{22})
$C_{21} =$ **Matrix-Multiply-DC** (A_{21}, B_{11}) + **Matrix-Multiply-DC** (A_{22}, B_{21})
$C_{22} =$ **Matrix-Multiply-DC** (A_{21}, B_{12}) + **Matrix-Multiply-DC** (A_{22}, B_{22})

return C

Running Time

Matrix-Multiply-DC (A, B)

if $n = 1$, **do**

$c_{11} = a_{11} \cdot b_{11}$

return c_{11}

Partition $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ **and** $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

$C_{11} =$ **Matrix-Multiply-DC** (A_{11}, B_{11}) + **Matrix-Multiply-DC** (A_{12}, B_{21})
$C_{12} =$ **Matrix-Multiply-DC** (A_{11}, B_{12}) + **Matrix-Multiply-DC** (A_{12}, B_{22})
$C_{21} =$ **Matrix-Multiply-DC** (A_{21}, B_{11}) + **Matrix-Multiply-DC** (A_{22}, B_{21})
$C_{22} =$ **Matrix-Multiply-DC** (A_{21}, B_{12}) + **Matrix-Multiply-DC** (A_{22}, B_{22})

return C

Running Time

Matrix-Multiply-DC (A, B)

if $n = 1$, **do**

$c_{11} = a_{11} \cdot b_{11}$ $\Theta(1)$

return c_{11}

Partition $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ **and** $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

$C_{11} =$ **Matrix-Multiply-DC** (A_{11}, B_{11}) + **Matrix-Multiply-DC** (A_{12}, B_{21})
$C_{12} =$ **Matrix-Multiply-DC** (A_{11}, B_{12}) + **Matrix-Multiply-DC** (A_{12}, B_{22})
$C_{21} =$ **Matrix-Multiply-DC** (A_{21}, B_{11}) + **Matrix-Multiply-DC** (A_{22}, B_{21})
$C_{22} =$ **Matrix-Multiply-DC** (A_{21}, B_{12}) + **Matrix-Multiply-DC** (A_{22}, B_{22})

return C

Running Time

Matrix-Multiply-DC (A, B)

if $n = 1$, **do**

$c_{11} = a_{11} \cdot b_{11}$ $\Theta(1)$

return c_{11}

Partition $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ **and** $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$ CLRS pp 82-83

$\Theta(1)$

$C_{11} =$ **Matrix-Multiply-DC** (A_{11}, B_{11}) + **Matrix-Multiply-DC** (A_{12}, B_{21})
$C_{12} =$ **Matrix-Multiply-DC** (A_{11}, B_{12}) + **Matrix-Multiply-DC** (A_{12}, B_{22})
$C_{21} =$ **Matrix-Multiply-DC** (A_{21}, B_{11}) + **Matrix-Multiply-DC** (A_{22}, B_{21})
$C_{22} =$ **Matrix-Multiply-DC** (A_{21}, B_{12}) + **Matrix-Multiply-DC** (A_{22}, B_{22})

return C

Running Time

Matrix-Multiply-DC (A, B)

if $n = 1$, **do**

$c_{11} = a_{11} \cdot b_{11}$ $\Theta(1)$

return c_{11}

Partition $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ **and** $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$ CLRS pp 82-83 $\Theta(1)$

$C_{11} =$ **Matrix-Multiply-DC** (A_{11}, B_{11}) + **Matrix-Multiply-DC** (A_{12}, B_{21})
$C_{12} =$ **Matrix-Multiply-DC** (A_{11}, B_{12}) + **Matrix-Multiply-DC** (A_{12}, B_{22})
$C_{21} =$ **Matrix-Multiply-DC** (A_{21}, B_{11}) + **Matrix-Multiply-DC** (A_{22}, B_{21})
$C_{22} =$ **Matrix-Multiply-DC** (A_{21}, B_{12}) + **Matrix-Multiply-DC** (A_{22}, B_{22})

return C

addition: $\Theta(n^2)$

Running Time

Matrix-Multiply-DC (A, B)

if $n = 1$, **do**

$c_{11} = a_{11} \cdot b_{11}$ $\Theta(1)$

return c_{11}

Partition $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ **and** $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$ CLRS pp 82-83 $\Theta(1)$

$C_{11} =$ **Matrix-Multiply-DC** (A_{11}, B_{11}) + **Matrix-Multiply-DC** (A_{12}, B_{21})

$C_{12} =$ **Matrix-Multiply-DC** (A_{11}, B_{12}) + **Matrix-Multiply-DC** (A_{12}, B_{22}) $8T(n/2)$

$C_{21} =$ **Matrix-Multiply-DC** (A_{21}, B_{11}) + **Matrix-Multiply-DC** (A_{22}, B_{21})

$C_{22} =$ **Matrix-Multiply-DC** (A_{21}, B_{12}) + **Matrix-Multiply-DC** (A_{22}, B_{22})

return C

addition: $\Theta(n^2)$

Running Time

Recurrence relation: For some constant c,

$$T(n) = \begin{cases} 8T(n/2) + cn^2, \text{ when } n > 1. \\ c, \text{ when } n = 1. \end{cases}$$

The Master Theorem

The Master Theorem is a very general theorem for solving recurrence relations.

Suppose $T(n) \leq \alpha T\left(\lceil n/b \rceil\right) + O(n^d)$

for some constants $\alpha > 0,\ b > 1$ and $d \geq 0$.

$$\text{Then,}\quad T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b \alpha \\ O(n^d \log_b n), & \text{if } d = \log_b \alpha \\ O(n^{\log_b \alpha}), & \text{if } d < \log_b \alpha \end{cases}$$

The Master Theorem

The Master Theorem is a very general theorem for solving recurrence relations.

Suppose $T(n) \leq \alpha T\left(\lceil n/b \rceil\right) + O(n^d)$
for some constants $\alpha > 0,\ b > 1$ and $d \geq 0$.

$$\text{Then,}\quad T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b \alpha \\ O(n^d \log_b n), & \text{if } d = \log_b \alpha \\ O(n^{\log_b \alpha}), & \text{if } d < \log_b \alpha \end{cases}$$

For **Matrix-Multiply-DC**, $\alpha = 8$, $b = 2$ and $d = 2$, we get $O(n^3)$

A whole in the water

For **Matrix-Multiply-DC**, $\alpha = 8,\ b = 2$ and $d = 2$, we get $O(n^3)$

A Greek idiom that means that despite trying, we didn't manage to achieve anything useful.

A whole in the water

For **Matrix-Multiply-DC**, $\alpha = 8$, $b = 2$ and $d = 2$, we get $O(n^3)$

A Greek idiom that means that despite trying, we didn't manage to achieve anything useful.

A whole in the water

For **Matrix-Multiply-DC**, $\alpha = 8$, $b = 2$ and $d = 2$, we get $O(n^3)$

A Greek idiom that means that despite trying, we didn't manage to achieve anything useful.

Strassen's remarkable algorithm to the rescue (next lecture)