W03: Lower Bounds and Asymptotic Analysis

CS3230 AY21/22 Sem 2

Click on the link to jump to the relevant sections!

Table of Contents

Question 1

Given an **unsorted array** of *n* real numbers ***`A[1...n]`*** and a query number ***`x`***. Develop a `search(x, A)` which returns an integer **`i`** if **`A[i]=x`**, and returns `-1` otherwise

Assumptions:

- Comparison Model
- Each comparison returns <, or > or = between *x* and an element of A

What is the lower bound on the **number of comparisons**?

Question 1

`Search(24, A) = ?`

Given an **unsorted array** of *n* real numbers ***A[1...n]*** and a query number ***x***. Develop a `search(x, A)` which returns an integer **`i`** if **`A[i]=x`**, and returns `-1` otherwise

A

18	2	3	5	6	24	23
1	2	3	4	5	6	7

Question 1

`Search(24, A) = 6`

Given an **unsorted array** of *n* real numbers *`A[1...n]`* and a query number *`x`*. Develop a `search(x, A)` which returns an integer **`i`** if **`A[i]=x`**, and returns `-1` otherwise

A	18	2	3	5	6	24	23
	1	2	3	4	5	6	7

Question 1

`Search(100, A) = ?`

Given an **unsorted array** of *n* real numbers ***A[1...n]*** and a query number ***x***. Develop a `search(x, A)` which returns an integer **`i`** if **`A[i]=x`**, and returns `-1` otherwise

A	18	2	3	5	6	24	23
	1	2	3	4	5	6	7

Question 1

```
Search(100, A) = -1
Couldn’t find anything!
```

Given an **unsorted array** of *n* real numbers ***A[1...n]*** and a query number ***x***. Develop a `search(x, A)` which returns an integer **`i`** if **`A[i]=x`**, and returns `-1` otherwise

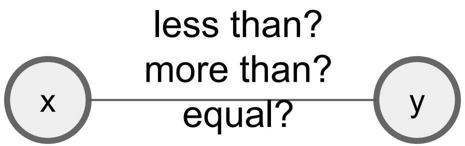
A	18	2	3	5	6	24	23
	1	2	3	4	5	6	7

Question 1 (Solution)

Given an **unsorted array** of n real numbers $A[1...n]$ and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Ans: The lower bound is n

Idea: A **comparison** tells us the **relationship** between **two numbers**

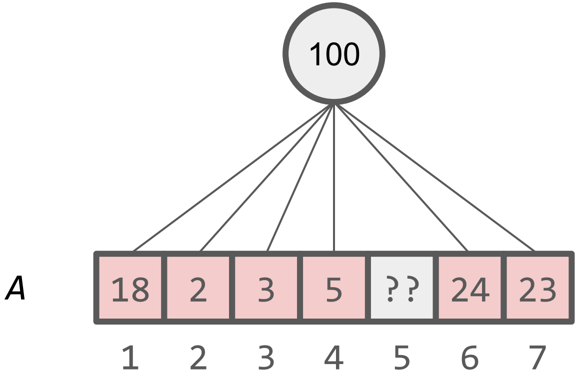


Question 1 (Solution)

Given an **unsorted array** of n real numbers `A[1...n]` and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Proof (by contradiction): Assume lower bound is not n. We can have an algorithm that solves it in $n-1$ comparisons

- We only know the relationship between x and at most $n-1$ elements

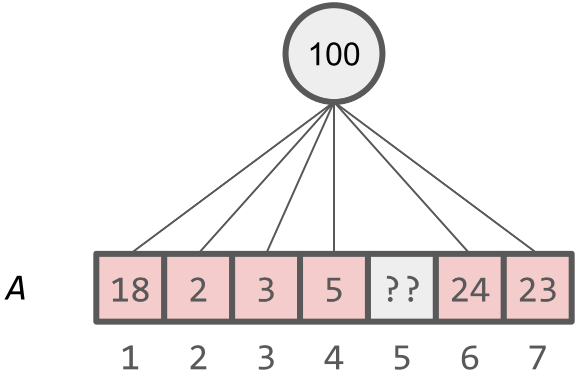


Question 1 (Solution)

Given an **unsorted array** of n real numbers `A[1...n]` and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Proof (by contradiction): Assume lower bound is not n. We can have an algorithm that solves it in $n-1$ comparisons

- We only know the relationship between x and at most $n-1$ elements

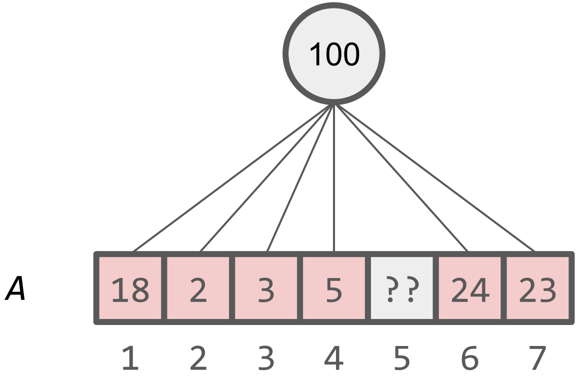


Question 1 (Solution)

Given an **unsorted array** of n real numbers `A[1...n]` and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

An adversary came along! He creates two arrays "almost identical" to A:

- *B* where *B[5] = 100*
- *C* where *C[5] ≠ 100*



Question 1 (Solution)

Given an **unsorted array** of *n* real numbers ***A[1...n]*** and a query number ***x***. Develop a `search(x, A)` which returns an integer **i** if **A[i]=x**, and returns **-1** otherwise

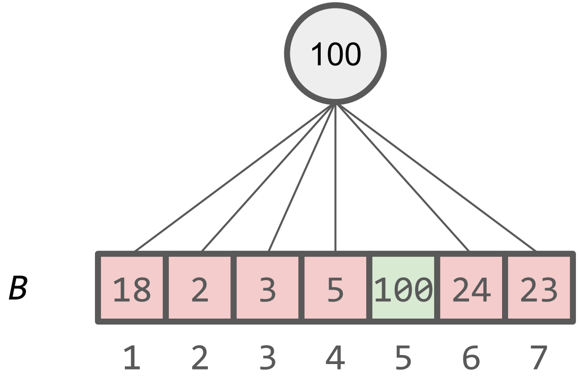
An adversary came along! He creates two arrays “almost identical” to A:

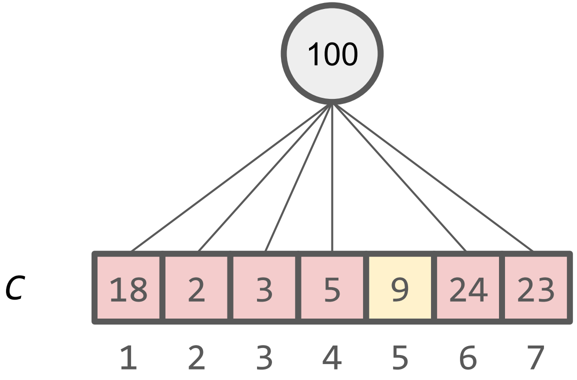
- *B* where *B[5] = 100*
- *C* where *C[5] ≠ 100*

Question 1 (Solution)

Given an **unsorted array** of *n* real numbers `A[1...n]` and a query number *x*. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

- The algorithm (with n - 1) comparisons will now return the same output for array B and C (because it cannot differentiate the two array)

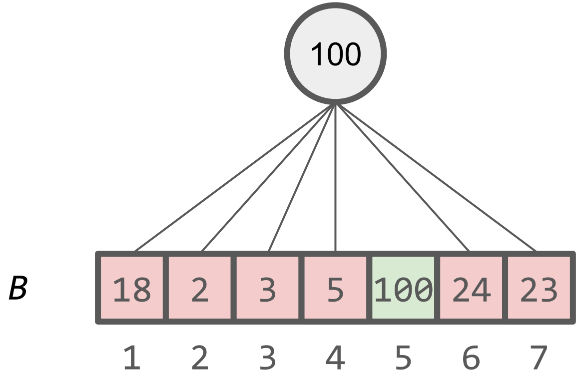


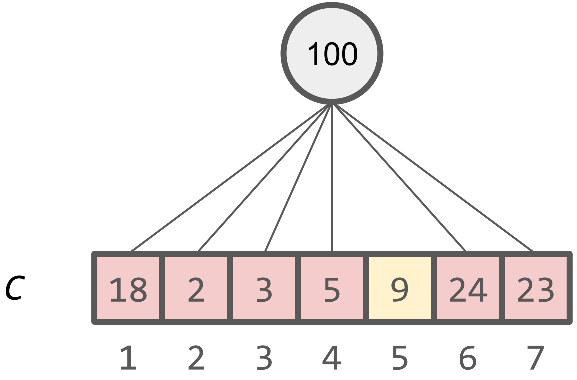


Question 1 (Solution)

Given an **unsorted array** of *n* real numbers ***A[1...n]*** and a query number ***x***. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

- The algorithm (with n - 1) comparisons will now return the same output for array B and C (because it cannot differentiate the two array)
- But both searching on B and C **should have different solutions**. Contradiction!





Question 1 (Solution)

Minor additional detail:

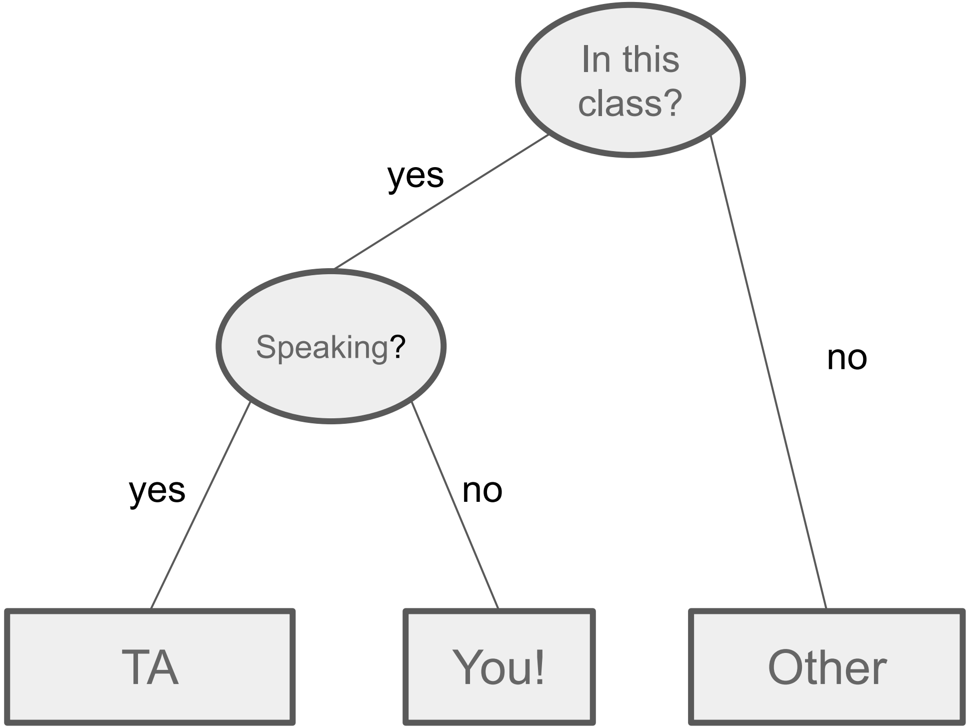
The adversary needs to first "answer" n - 1 queries with fixed values to find out which position was not queried. This is because the unqueried position could depend on the answers to previous queries.

After that, the adversary can construct the two indistinguishable arrays

Decision Tree

Decision Tree

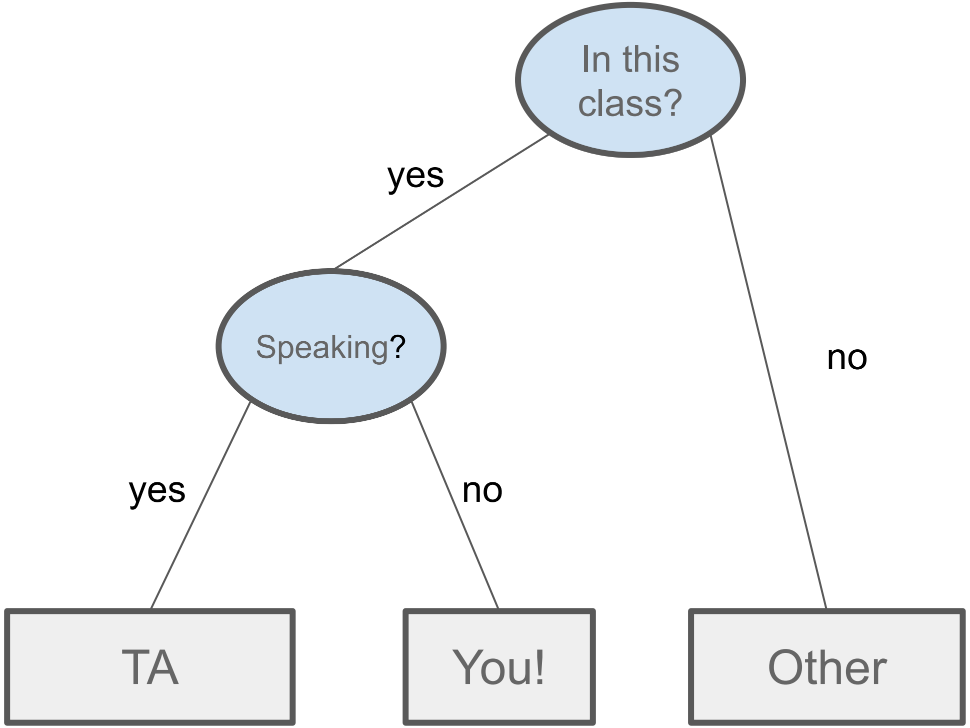
A **decision tree** is a tree-like model



Decision Tree

A **decision tree** is a tree-like model

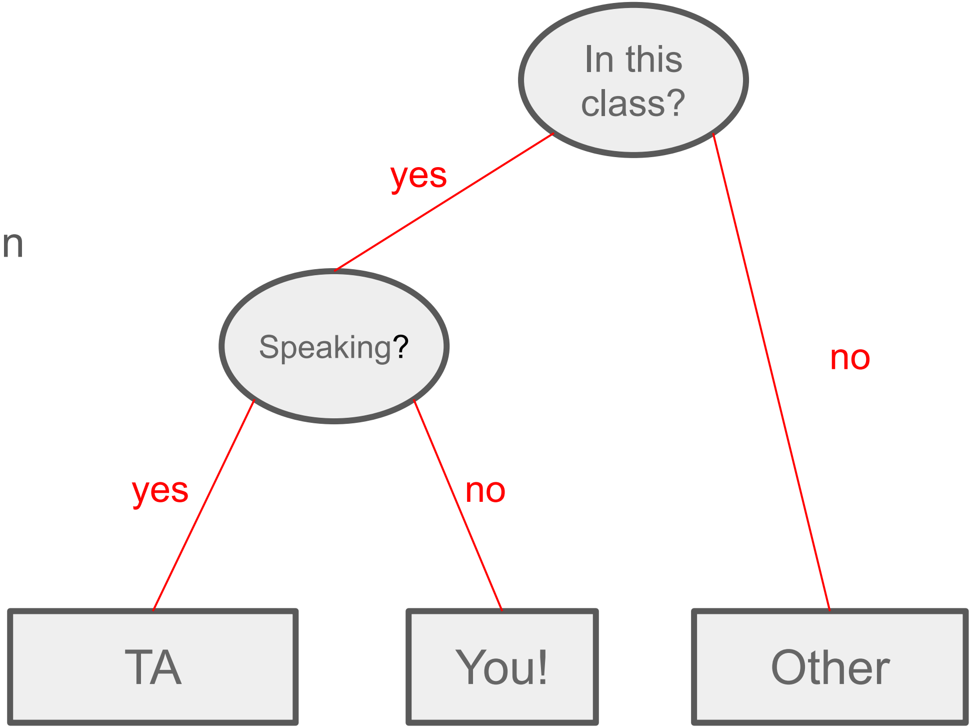
- A node is a **comparison**



Decision Tree

A **decision tree** is a tree-like model

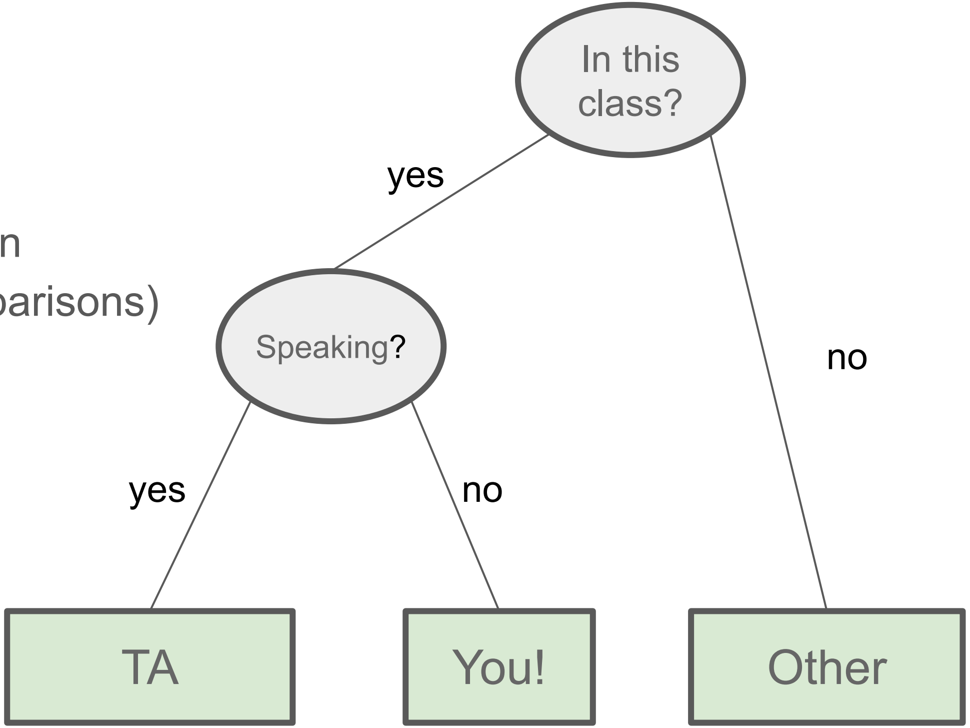
- A node is a **comparison**
- A branch is the **outcome** of comparison



Decision Tree

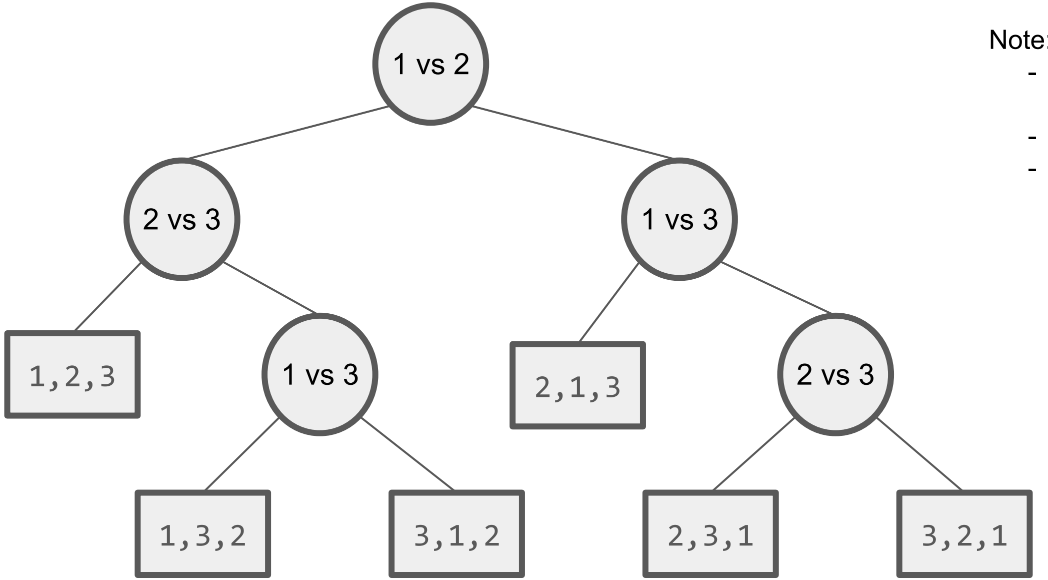
A **decision tree** is a tree-like model

- A node is a **comparison**
- A branch is the **outcome** of comparison
- A leaf is a label (decision after all comparisons)



Decision Tree

Sorting Decision trees!

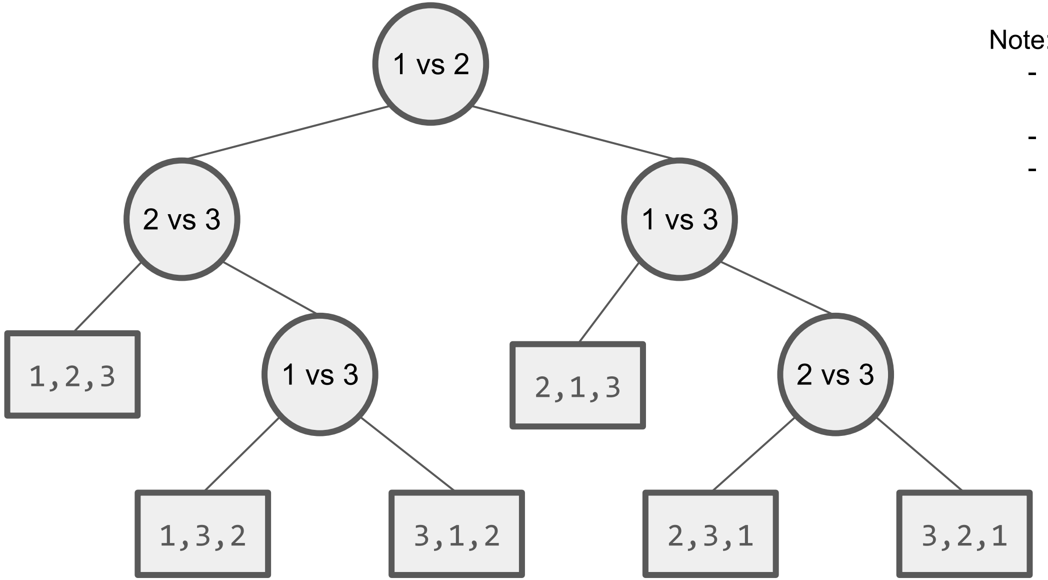


Note:

- i vs j means compare at index i and index j
- going left means “less than”
- going right means “more than”

Decision Tree

```
Want to sort 9 4 6
```

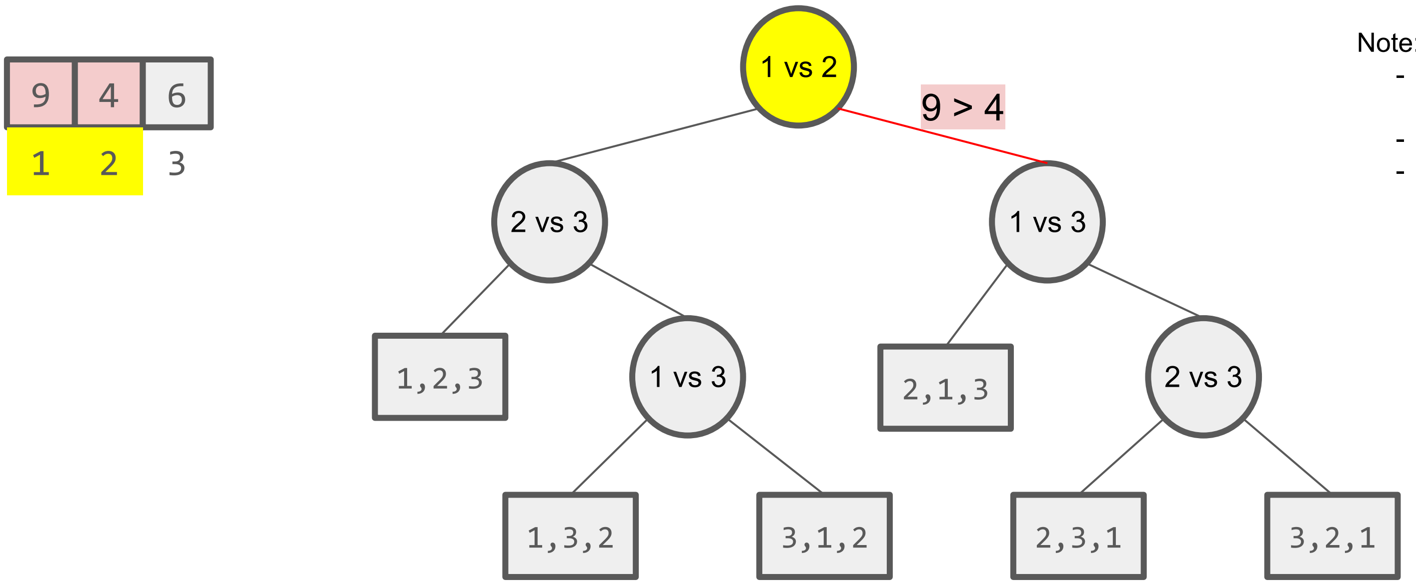


Note:

- i vs j means compare at index i and index j
- going left means “less than”
- going right means “more than”

Decision Tree

Compare at idx 1 and idx 2

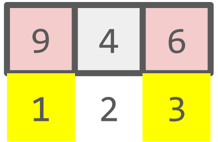


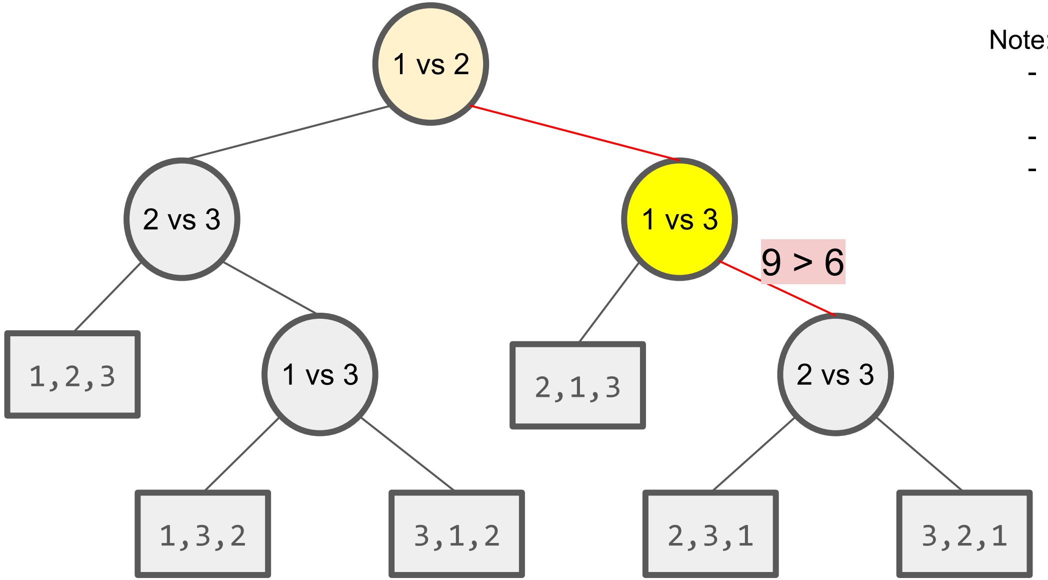
Note:

- i vs j means compare at index i and index j
- going left means “less than”
- going right means “more than”

Decision Tree

Compare at idx 1 and idx 3



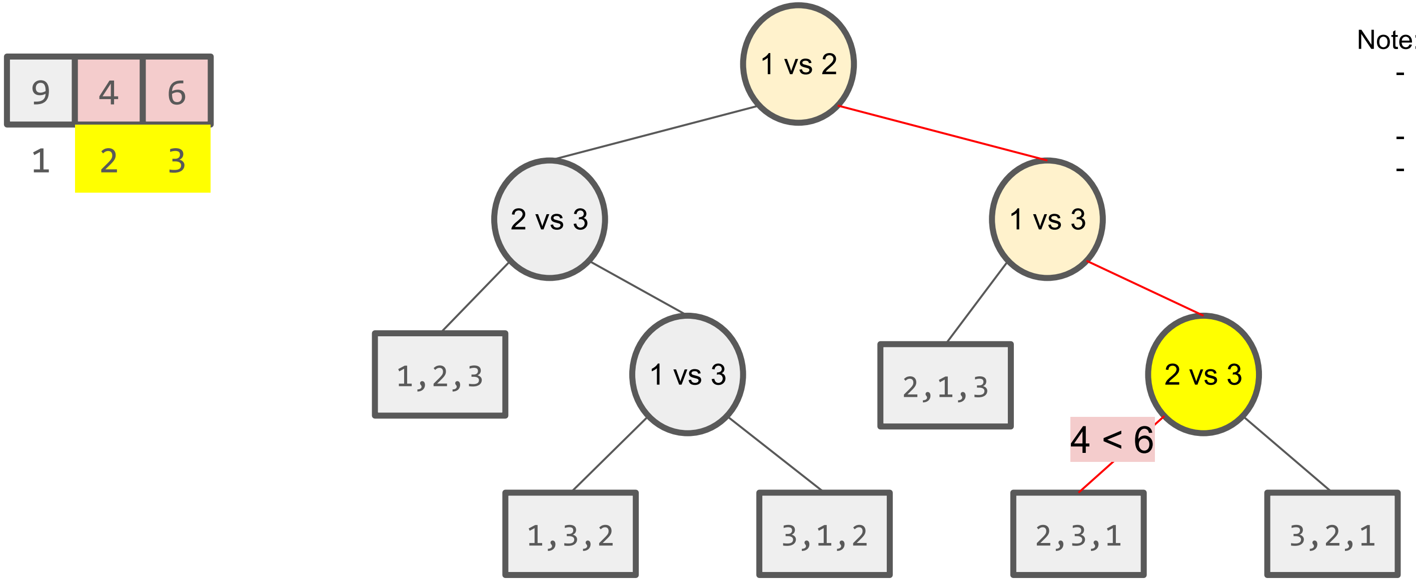


Note:

- i vs j means compare at index i and index j
- going left means “less than”
- going right means “more than”

Decision Tree

Compare at idx 2 and idx 3



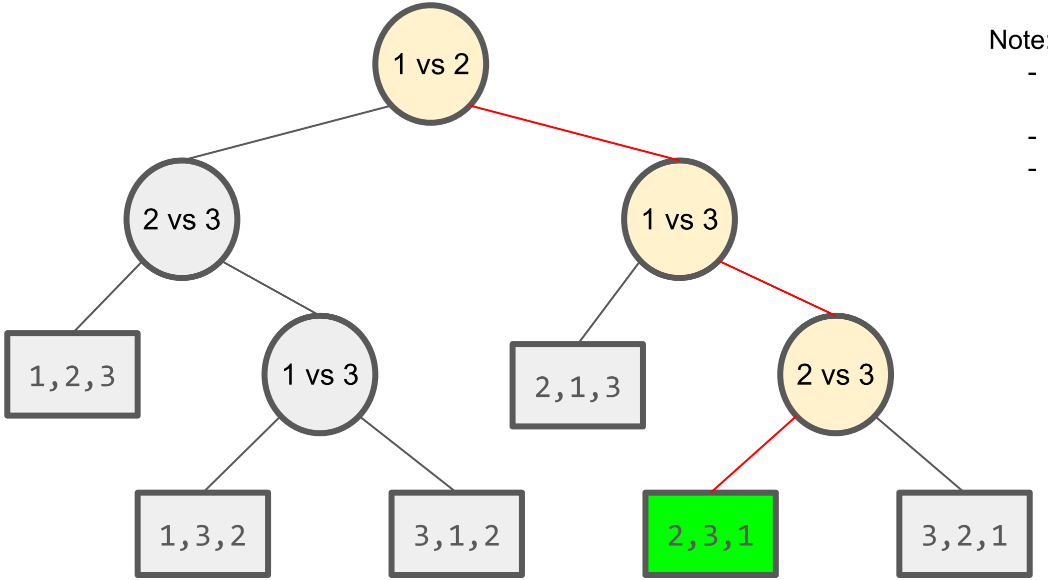
Note:

- i vs j means compare at index i and index j
- going left means "less than"
- going right means "more than"

Decision Tree

We conclude that the indices must appear in order:
2, 3, 1

9	4	6
1	2	3

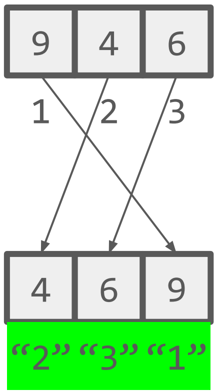


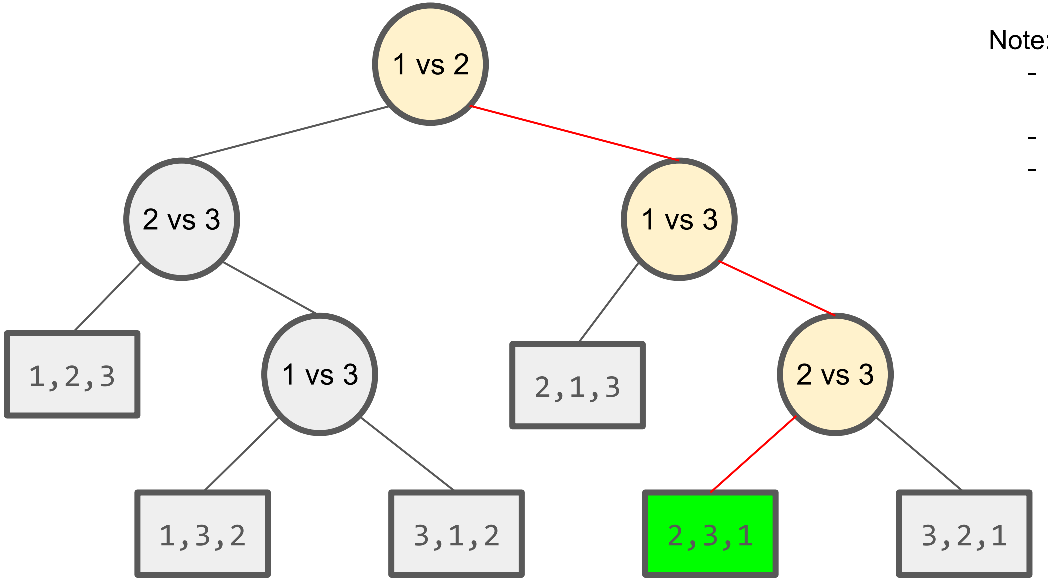
Note:

- i vs j means compare at index i and index j
- going left means “less than”
- going right means “more than”

Decision Tree

Verify that it is indeed sorted!





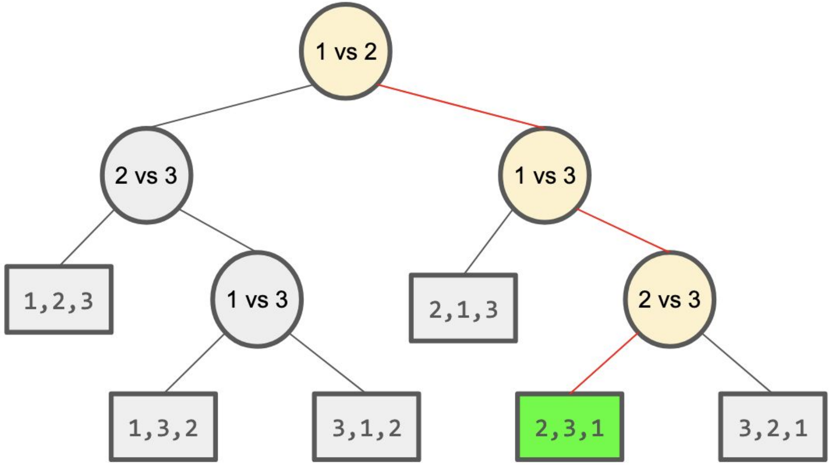
Note:

- i vs j means compare at index i and index j
- going left means "less than"
- going right means "more than"

Decision Tree

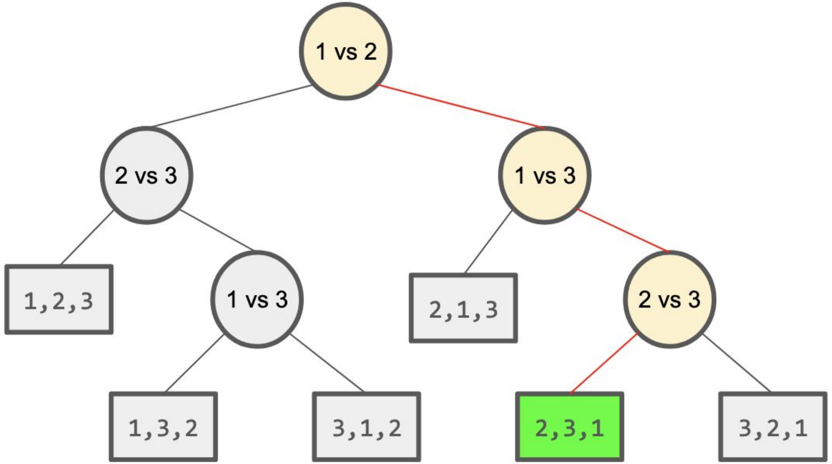
Decision Tree **models** execution of any comparison sort:

- One tree for each n (i.e. different trees if n=3, n=4, etc)
- View the algorithm as “splitting” whenever a comparison is made



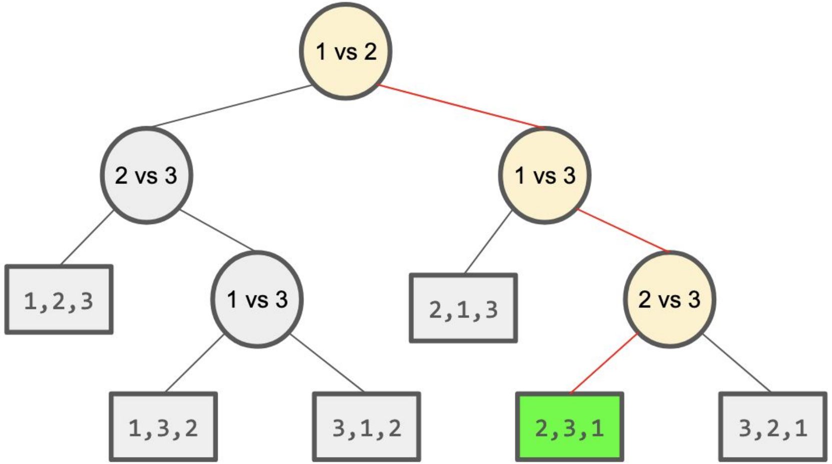
Decision Tree and runtime

- Runtime of algorithm = length of path taken (in this example can be 2 or 3)



Decision Tree and runtime

- Runtime of algorithm = length of path taken (in this example can be 2 or 3)
- Worst-case running time = height of the tree (in this example it's 3)



Decision Tree Sorting

Theorem: Any decision tree that can sort n elements must have height $\Omega(n\lg n)$

Decision Tree Sorting

Theorem: Any decision tree that can sort n elements must have height $\Omega(n\lg n)$

Claim 1: A height h binary tree has $\leq 2^h$ leaves

Decision Tree Sorting

Theorem: Any decision tree that can sort *n* elements must have height $\Omega(n\lg n)$

Claim 1: A height *h* binary tree has ≤ 2^h leaves

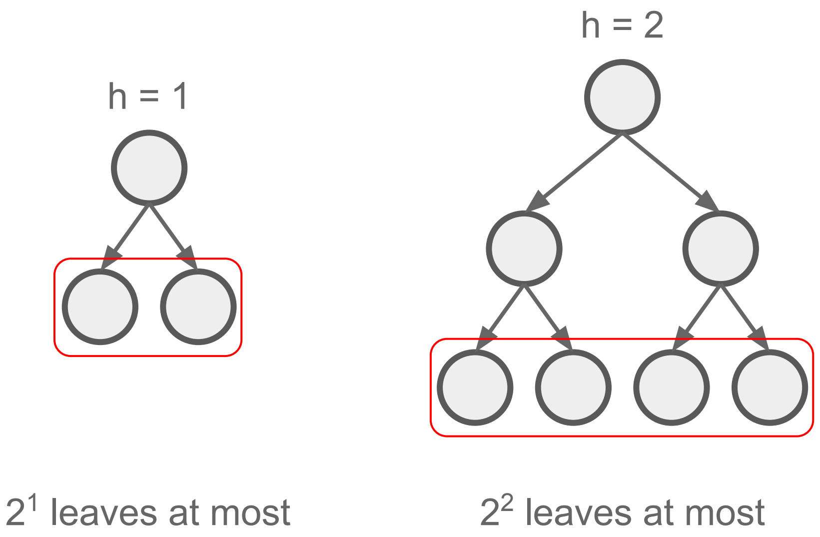
Proof: Can be done by using mathematical induction on the height (exercise!)

Decision Tree Sorting

Theorem: Any decision tree that can sort *n* elements must have height *Ω(nlgn)*

Claim 1: A height *h* binary tree has $\leq 2^h$ leaves

Visual Idea instead of Proof:



Decision Tree Sorting

Theorem: Any decision tree that can sort n elements must have height $\Omega(n\lg n)$

Claim 2: The decision tree must contain $n!$ leaves

Decision Tree Sorting

Theorem: Any decision tree that can sort *n* elements must have height $\Omega(n\lg n)$

Claim 2: The decision tree must contain *n!* leaves

Proof:

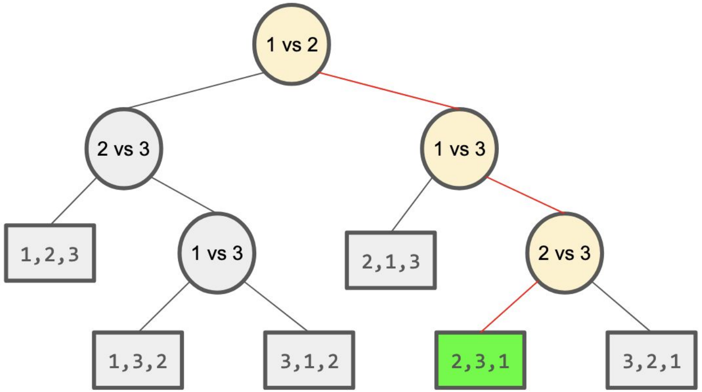
- The outcome of the sorting can be *any* permutation of the input array
- There are *n!* permutations → there are *n!* leaves

Theorem: Any decision tree that can sort *n* elements must have height *Ω(nlgn)*

Claim 1: A height *h* binary tree has ≤ 2^h leaves

Claim 2: The decision tree must contain *n!* leaves

Recall: Worst-case running time is the **height** of the decision tree.



Theorem: Any decision tree that can sort *n* elements must have height *Ω(nlgn)*

Claim 1: A height *h* binary tree has ≤ 2^h leaves

Claim 2: The decision tree must contain *n!* leaves

Recall: Worst-case running time is the **height** of the decision tree.

Ask ourselves: We have *n!* leaves. What’s our *minimum* height? How to relate n and h?

Theorem: Any decision tree that can sort *n* elements must have height *Ω(nlgn)*

Claim 1: A height *h* binary tree has $\leq 2^h$ leaves
Claim 2: The decision tree must contain *n!* leaves

Recall: Worst-case running time is the **height** of the decision tree.

Ask ourselves: We have *n!* leaves. What's our *minimum* height? How to relate n and h? It's $n! \leq 2^h$

Theorem: Any decision tree that can sort *n* elements must have height *Ω(nlgn)*

Claim 1: A height *h* binary tree has $\leq 2^h$ leaves
Claim 2: The decision tree must contain *n!* leaves

Recall: Worst-case running time is the **height** of the decision tree.

Ask ourselves: We have *n!* leaves. What's our *minimum* height? How to relate n and h? It's $n! \leq 2^h$

$h \geq \lg(n!)$ (lg is monotonically increasing)

Theorem: Any decision tree that can sort *n* elements must have height $\Omega(n \lg n)$

Claim 1: A height *h* binary tree has $\leq 2^h$ leaves
Claim 2: The decision tree must contain *n!* leaves

Recall: Worst-case running time is the **height** of the decision tree.

Ask ourselves: We have *n!* leaves. What's our *minimum* height? How to relate n and h? It's $n! \leq 2^h$

$$\begin{aligned} h &\geq \lg(\boxed{n!}) \quad \text{(lg is monotonically increasing)} \\ &\geq \lg(\boxed{(n/e)^n}) \quad \text{(Stirling's formula)} \end{aligned}$$

$$\boxed{n!} = \sqrt{2\pi n}\,\boxed{\left(\frac{n}{e}\right)^n}$$

Theorem: Any decision tree that can sort *n* elements must have height $\Omega(n\lg n)$

Claim 1: A height *h* binary tree has $\leq 2^h$ leaves
Claim 2: The decision tree must contain *n!* leaves

Recall: Worst-case running time is the **height** of the decision tree.

Ask ourselves: We have *n!* leaves. What's our *minimum* height? How to relate n and h? It's $n! \leq 2^h$

$$
\begin{aligned}
h &\geq \lg(n!) \quad \text{(lg is monotonically increasing)} \\
&\geq \lg((n/e)^n) \quad \text{(Stirling's formula)} \\
&= n \lg n - n \lg e \\
&= \Omega(n \lg n) . \quad \square
\end{aligned}
$$

Question 2

Given an **sorted array** of *n* real numbers ***`A[1...n]`*** and a query number ***`x`***. Develop a `search(x, A)` which returns an integer **`i`** if **`A[i]=x`**, and returns `-1` otherwise

Assumptions:

- Comparison Model
- Each comparison returns <, or > or = between *x* and an element of A

What is the lower bound on the **number of comparisons**?

Question 2

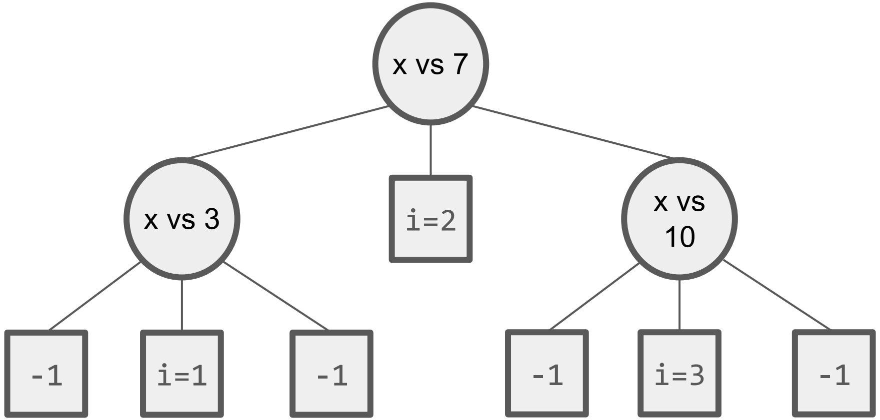
Given an **sorted array** of n real numbers `A[1...n]` and a query number x. Develop a `search(x, A)` which returns an integer **`i`** if **`A[i]=x`**, and returns `-1` otherwise

Assumptions:

- Comparison Model
- Each comparison returns $<$, or $>$ or $=$ between x and an element of A

What is the lower bound on the **number of comparisons**?

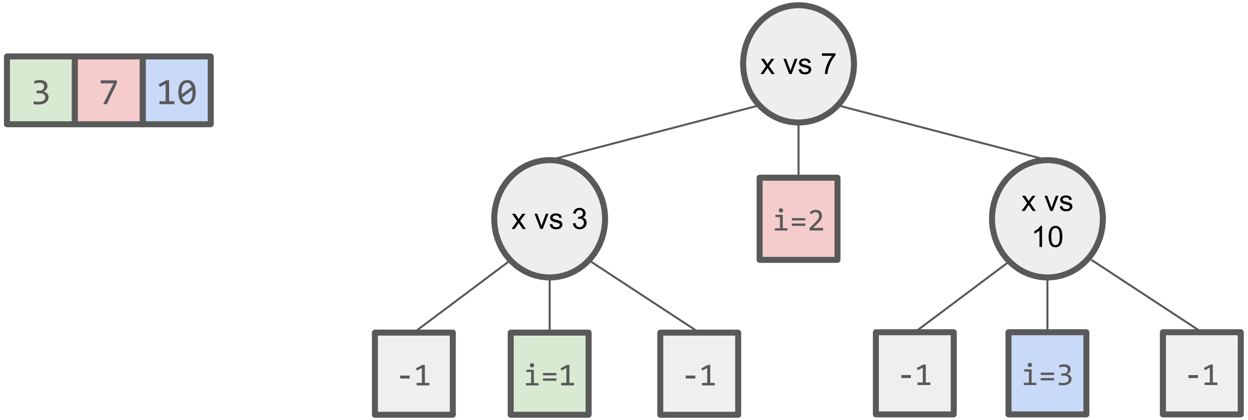
Question 2 Example: Lower bound is 2



Left = Less than
Center = Equal
Right = More than

Note that now we are comparing value instead of value at indices

Question 2 Example: Lower bound is 2



Left = Less than
Center = Equal
Right = More than

Note that now we are comparing value instead of value at indices

Coloured nodes: the “leaf” label to decide which of the position in the array

Question 2 (Solution)

Given an **sorted array** of n real numbers `A[1...n]` and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Answer: The lower bound is $\lfloor lg(n) \rfloor + 1$

Claim 1: A tree with height h has $\leq 2^{h+1} - 1$ nodes

Question 2 (Solution)

Given an **sorted array** of n real numbers `A[1...n]` and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Answer: The lower bound is $\lfloor lg(n) \rfloor + 1$

Claim 1: A tree with height h has $\leq 2^{h+1} - 1$ nodes

Proof: Can be done by using mathematical induction on the height (exercise!)

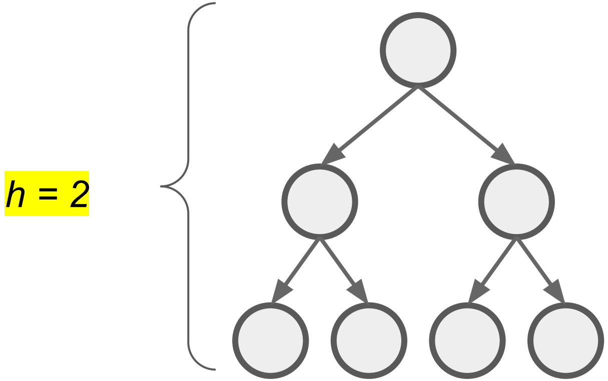
Question 2 (Solution)

Given an **sorted array** of *n* real numbers ***A[1...n]*** and a query number ***x***. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Answer: The lower bound is $\lfloor lg(n) \rfloor + 1$

Claim 1: A tree with height h has $\leq 2^{h+1} - 1$ nodes

Visual idea instead of induction:



Number of nodes:
$\boxed{2^0 + 2^1 + 2^2} = \boxed{2^3 - 1}$

Derived from sum of GP: $\boxed{\frac{a(r^n - 1)}{(r - 1)}} = \frac{1(2^3 - 1)}{(2 - 1)} = \boxed{2^3 - 1}$

Given an **sorted array** of n real numbers `A[1...n]` and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Answer: The lower bound is $\lfloor lg(n) \rfloor + 1$

Claim 1: A tree with height h has $\leq 2^{h+1} - 1$ nodes

Proof (by contradiction): Assume there exists a decision tree with height $\leq lg(n)$

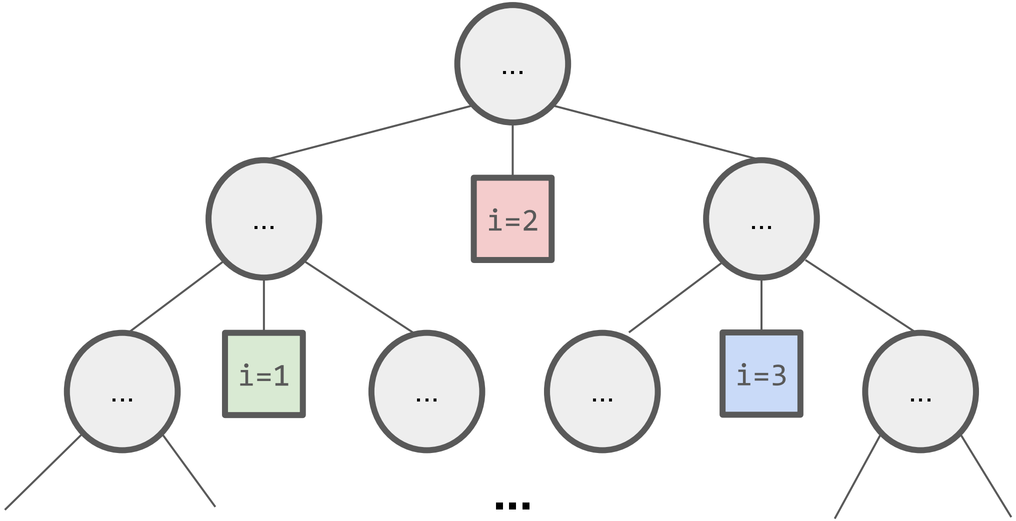
Given an **sorted array** of n real numbers `A[1...n]` and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Answer: The lower bound is $\lfloor lg(n) \rfloor + 1$

Claim 1: A tree with height h has ≤ $2^{h+1} - 1$ nodes

Proof (by contradiction): Assume there exists a decision tree with height ≤ $lg(n)$

Crux of the argument: If the tree height is too small, we are "missing out" some of the leaf labels (i.e. we cannot decide a certain index is the answer)



Answer: The lower bound is *⌊lg(n)⌋ + 1*

Given an **sorted array** of ***n*** real numbers ***A[1...n]*** and a query number ***x***. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Claim 1: A tree with height *h* has ≤ 2^{h+1} - 1 nodes

Crux of the argument: If the tree height is too small, we are "missing out" some of the leaf labels (i.e. we cannot decide a certain index is the answer)

Proof (by contradiction): Assume there exists a decision tree with height ≤ *lg(n)*

Idea: How many decisions (square nodes) that were part of our input array can we achieve with just *lg(n)* depth?

Given an **sorted array** of n real numbers `A[1...n]` and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

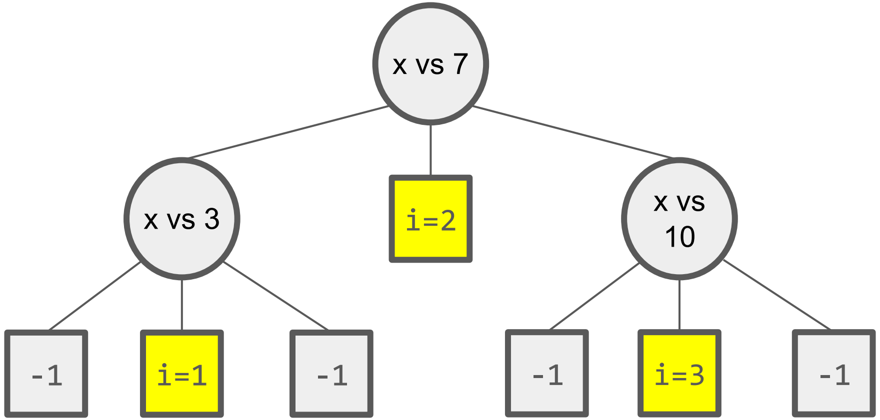
Answer: The lower bound is $\lfloor lg(n) \rfloor + 1$

Claim 1: A tree with height h has $\leq 2^{h+1} - 1$ nodes

Crux of the argument: If the tree height is too small, we are "missing out" some of the leaf labels (i.e. we cannot decide a certain index is the answer)

Proof (by contradiction): Assume there exists a decision tree with height $\leq lg(n)$

Idea: How many decisions (square nodes) that were part of our input array can we achieve with just $lg(n)$ depth?



Answer: The lower bound is $\lfloor lg(n) \rfloor + 1$

Given an **sorted array** of n real numbers `A[1...n]` and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

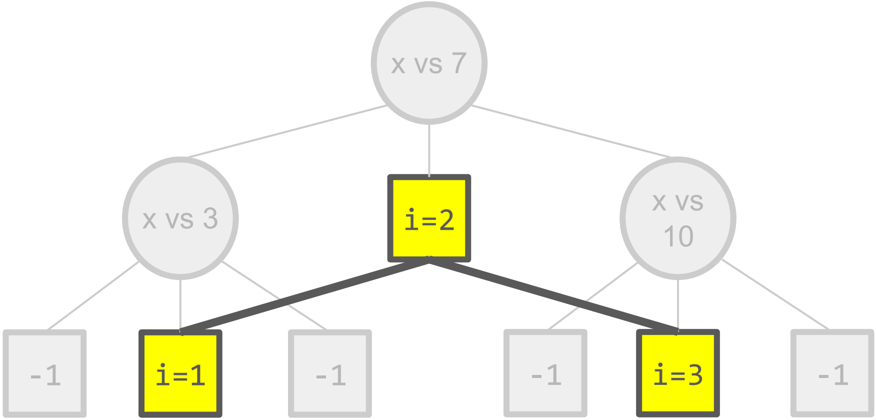
Claim 1: A tree with height h has $\leq 2^{h+1} - 1$ nodes

Crux of the argument: If the tree height is too small, we are "missing out" some of the leaf labels (i.e. we cannot decide a certain index is the answer)

Proof (by contradiction): Assume there exists a decision tree with height $\leq lg(n)$

Idea: How many decisions (square nodes) that were part of our input array can we achieve with just $lg(n)$ depth?

The internal square nodes form a little tree by itself! What is it height?



Answer: The lower bound is *⌊lg(n)⌋ + 1*

Given an **sorted array** of *n* real numbers *A[1...n]* and a query number *x*. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns -1 otherwise

Claim 1: A tree with height *h* has ≤ 2^{h+1} - 1 nodes

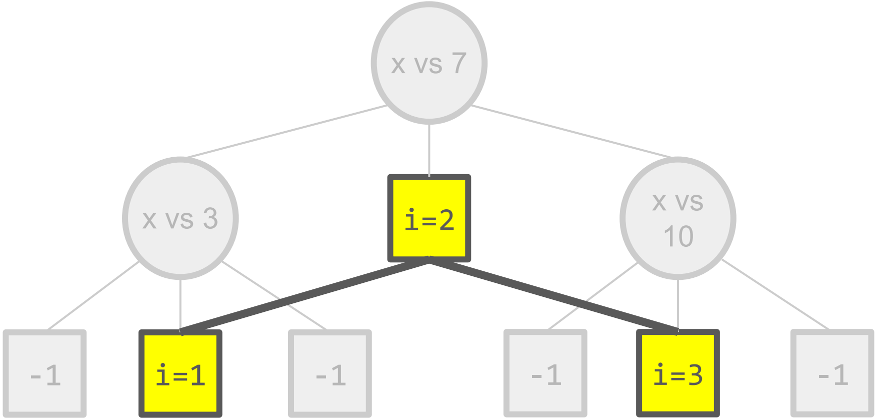
Crux of the argument: If the tree height is too small, we are "missing out" some of the leaf labels (i.e. we cannot decide a certain index is the answer)

Proof (by contradiction): Assume there exists a decision tree with height ≤ *lg(n)*

Idea: How many decisions (square nodes) that were part of our input array can we achieve with just *lg(n)* depth?

The internal square nodes form a little tree by itself! What is it height?

If original tree is height h, the little tree is height (h-1)



Given an **sorted array** of n real numbers $A[1...n]$ and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Answer: The lower bound is $\lfloor lg(n) \rfloor + 1$

Claim 1: A tree with height h has ≤ 2^{h+1} - 1 nodes

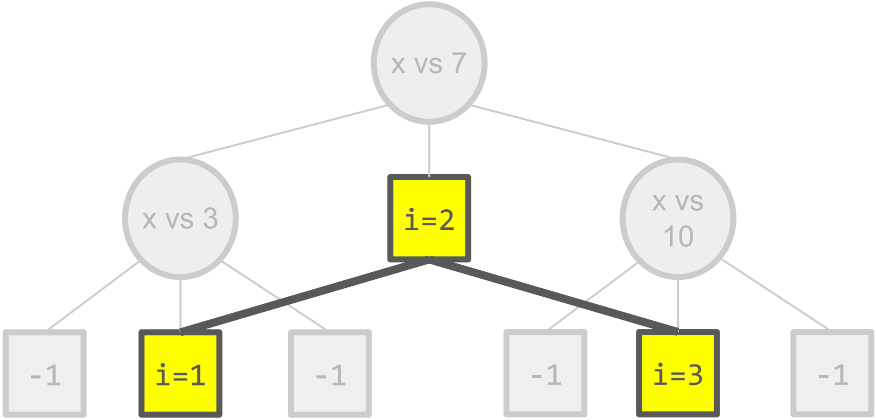
Crux of the argument: If the tree height is too small, we are "missing out" some of the leaf labels (i.e. we cannot decide a certain index is the answer)

Proof (by contradiction): Assume there exists a decision tree with height ≤ *lg(n)*

If the decision tree has height *lg(n)*, then the tree of internal nodes has height *lg(n)-1*

The internal square nodes form a little tree by itself! What is it height?

If original tree is height h, the little tree is height (h-1)



Answer: The lower bound is $\lfloor lg(n) \rfloor + 1$

Given an **sorted array** of n real numbers $A[1...n]$ and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

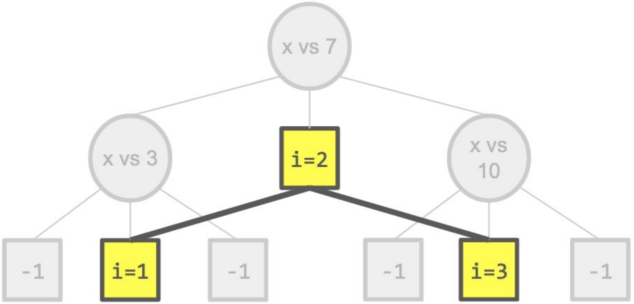
Claim 1: A tree with height h has $\leq 2^{h+1} - 1$ nodes

Crux of the argument: If the tree height is too small, we are "missing out" some of the leaf labels (i.e. we cannot decide a certain index is the answer)

Proof (by contradiction): Assume there exists a decision tree with height $\leq lg(n)$

If the decision tree has height $lg(n)$, then the tree of internal nodes has height $lg(n)-1$

By Claim 1:



Answer: The lower bound is *⌊lg(n)⌋ + 1*

Given an **sorted array** of n real numbers $A[1...n]$ and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Crux of the argument: If the tree height is too small, we are "missing out" some of the leaf labels (i.e. we cannot decide a certain index is the answer)

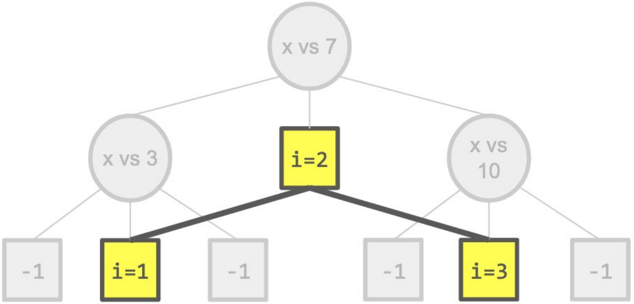
Claim 1: A tree with height h has ≤ 2^{h+1} - 1 nodes

Proof (by contradiction): Assume there exists a decision tree with height ≤ *lg(n)*

If the decision tree has height *lg(n)*, then the tree of internal nodes has height *lg(n)-1*

By Claim 1:

Tree with height *lg(n)-1* has ≤ $2^{lg(n)-1+1} - 1$ nodes



Answer: The lower bound is $\lfloor lg(n) \rfloor + 1$

Given an **sorted array** of n real numbers `A[1...n]` and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Claim 1: A tree with height h has $\leq 2^{h+1} - 1$ nodes

Crux of the argument: If the tree height is too small, we are "missing out" some of the leaf labels (i.e. we cannot decide a certain index is the answer)

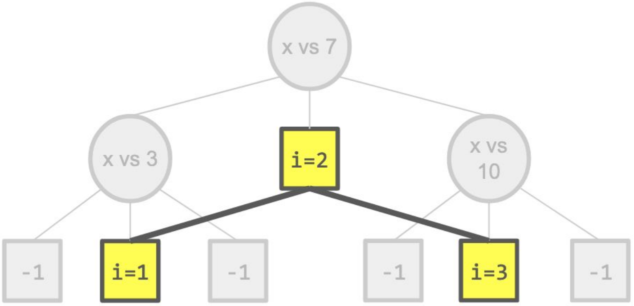
Proof (by contradiction): Assume there exists a decision tree with height $\leq lg(n)$

If the decision tree has height $lg(n)$, then the tree of internal nodes has height $lg(n)-1$

By Claim 1:

Tree with height $lg(n)-1$ has $\leq 2^{lg(n)-1+1} - 1$ nodes

$\leq n - 1$ nodes [because $2^{lg(n)} = n$]



Answer: The lower bound is $\lfloor lg(n) \rfloor + 1$

Given an **sorted array** of n real numbers `A[1...n]` and a query number x. Develop a `search(x, A)` which returns an integer `i` if `A[i]=x`, and returns `-1` otherwise

Crux of the argument: If the tree height is too small, we are "missing out" some of the leaf labels (i.e. we cannot decide a certain index is the answer)

Claim 1: A tree with height *h* has ≤ $2^{h+1} - 1$ nodes

Proof (by contradiction): Assume there exists a decision tree with height ≤ *lg(n)*

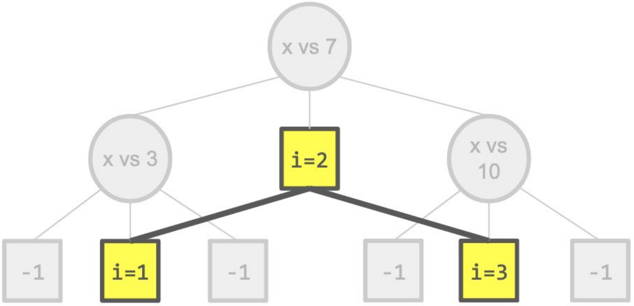
If the decision tree has height *lg(n)*, then the tree of internal nodes has height *lg(n)-1*

By Claim 1:

Tree with height *lg(n)-1* has ≤ $2^{lg(n)-1+1} - 1$ nodes

≤ n - 1 nodes [because $2^{lg(n)} = n$]

Contradiction because we should have started with n elements in the input!

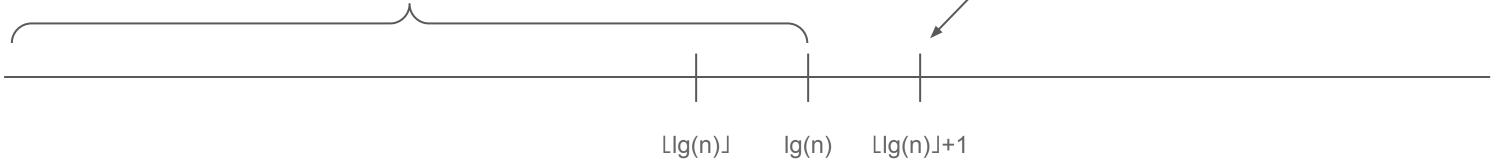


How to address the $\lfloor \lg(n) \rfloor$ in it?

The idea is that observe $\lfloor \lg(n) \rfloor \le \lg(n)$. So our proof of contradiction is claiming something stronger.

(1) We are showing if we do any number of comparisons in this range, then we derive a contradiction

(2) so the next **integer value** has to be the lower bound



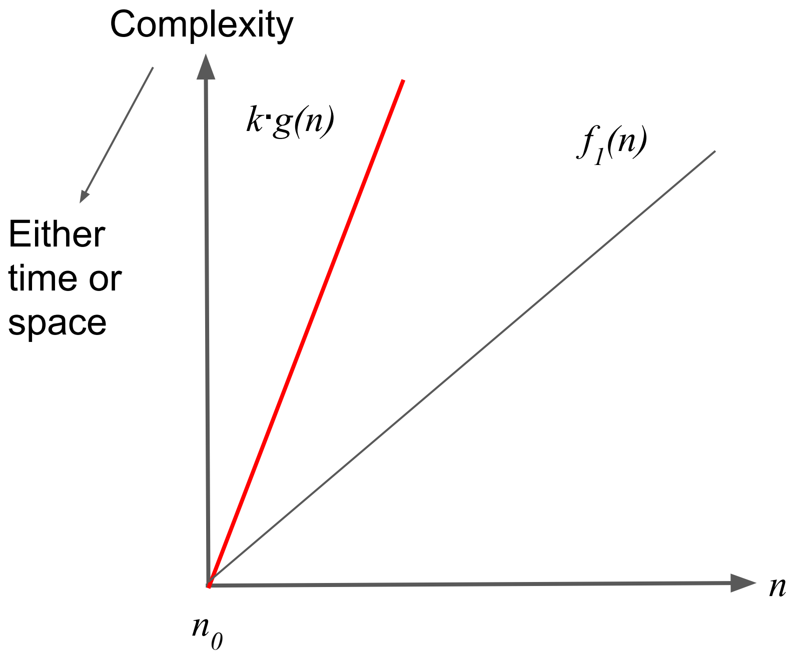
Additional notes for Q1 & Q2

Our arguments only show that it is impossible to have an algorithm with n-1 comparisons (Q1) and lg(n) comparisons (Q2).

To be more rigorous, we should also argue for the existence of algorithms that run in n and $\lfloor \lg(n) \rfloor + 1$ comparisons respectively, which has been omitted from the slides for brevity

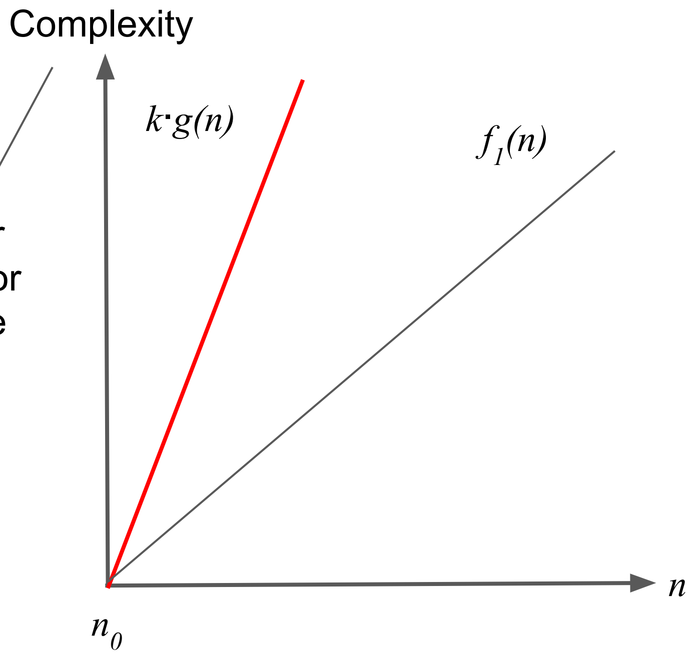
Asymptotic Analysis

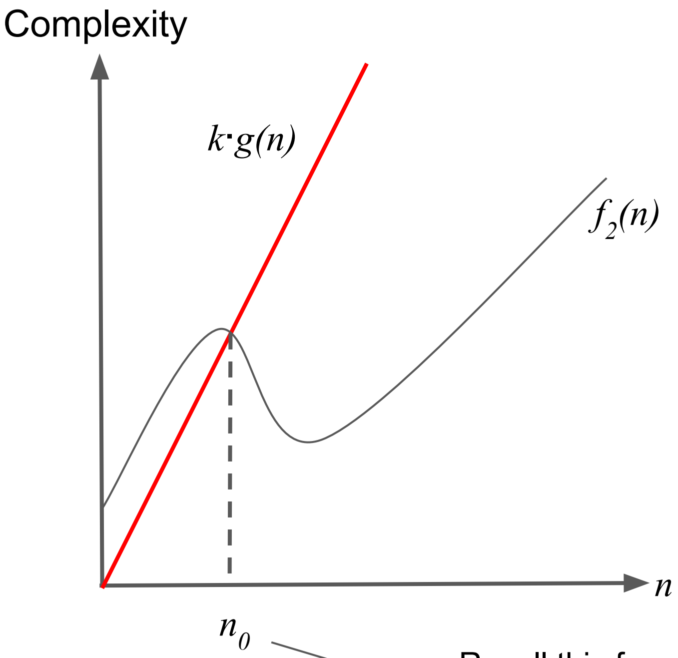
Big O (Upper bound) where $g(n) = n$



Big O (Upper bound) where $g(n) = n$

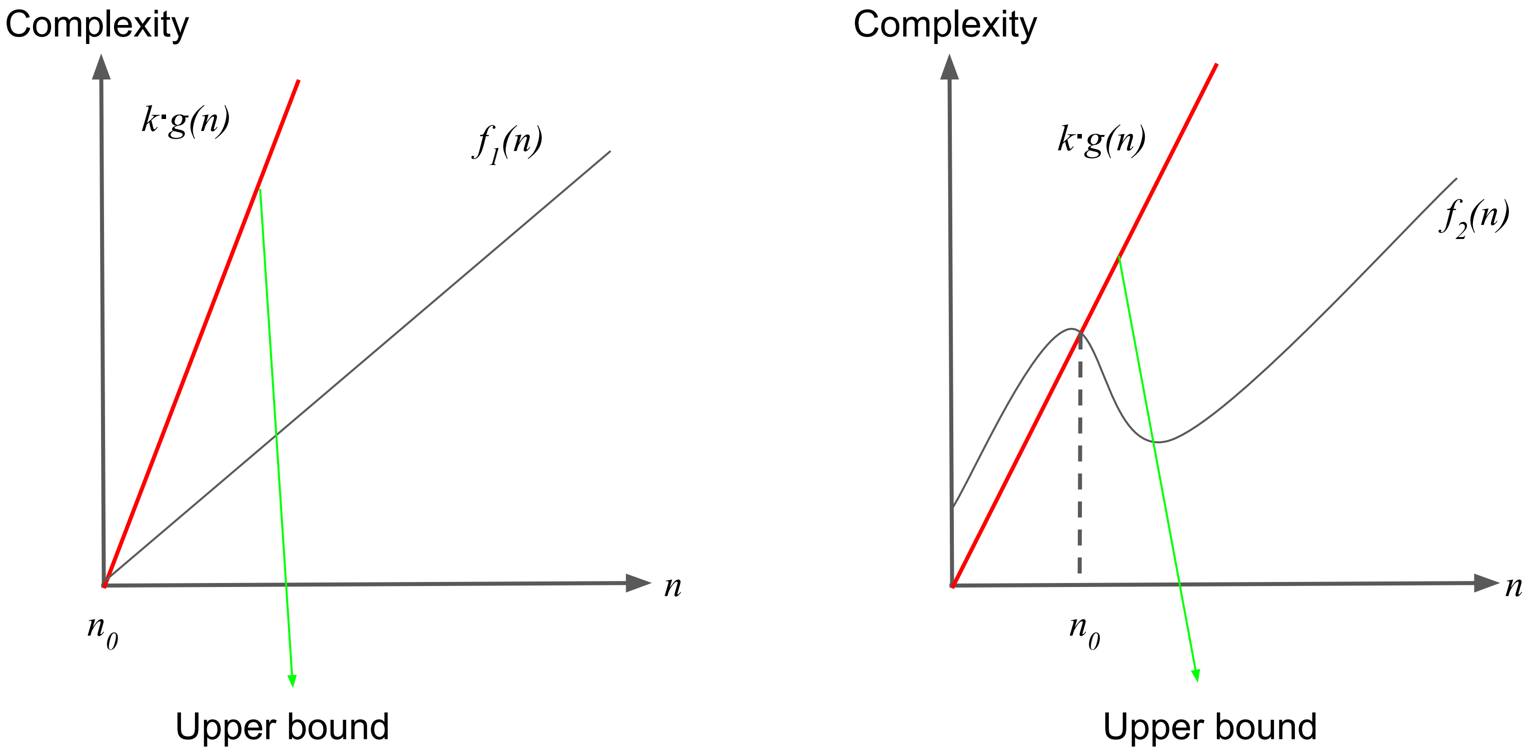
Either time or space



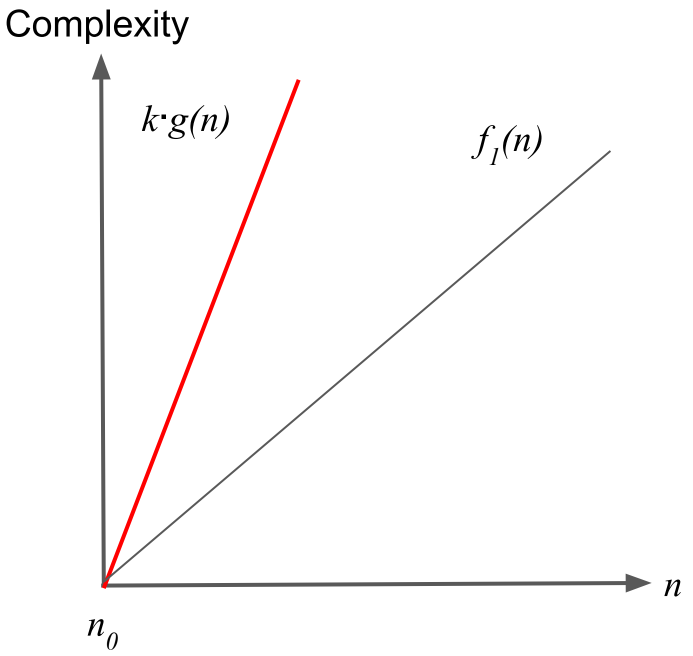


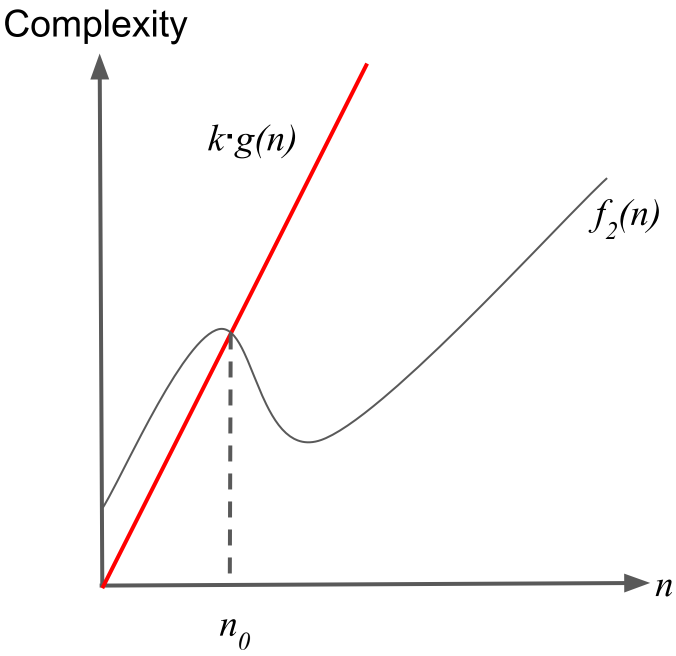
Recall this from the definition: “For any sufficiently large value of n”

Big O (Upper bound) where $g(n) = n$



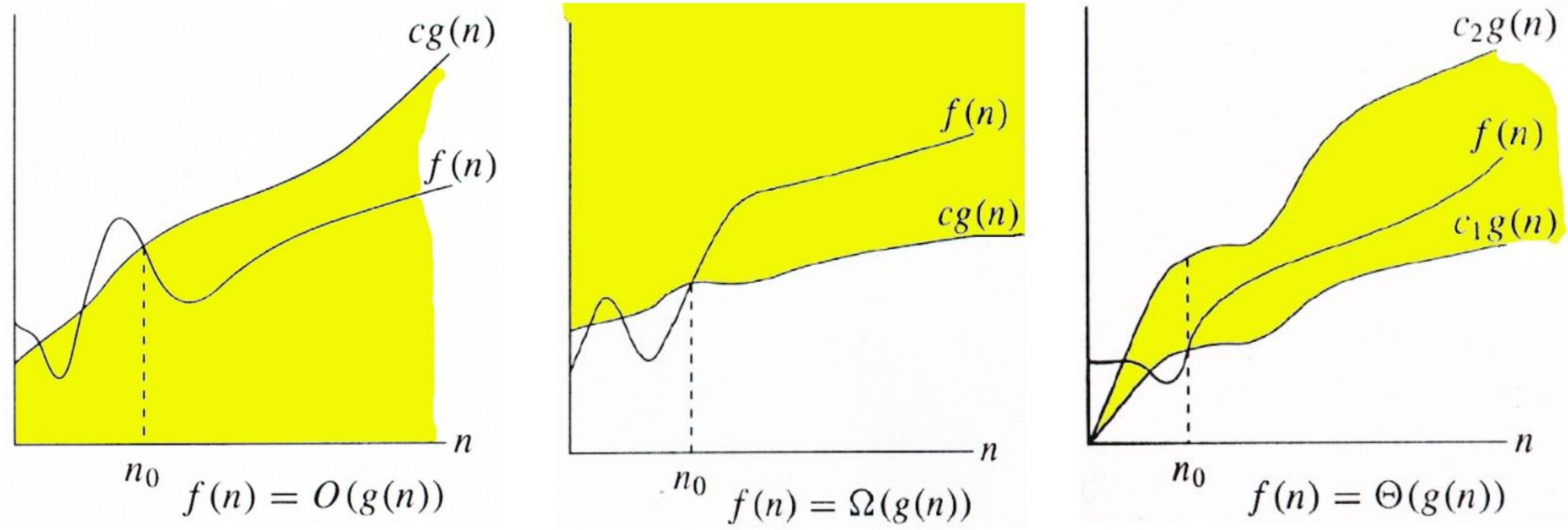
Big O (Upper bound) where $g(n) = n$





We can say that the functions f_1 and f_2, have order of growth of $O(g(n))$. More specifically, they have an order of growth of $\boldsymbol{O(n)}$.

The functions *f(n)* can be all sorts of funny things!

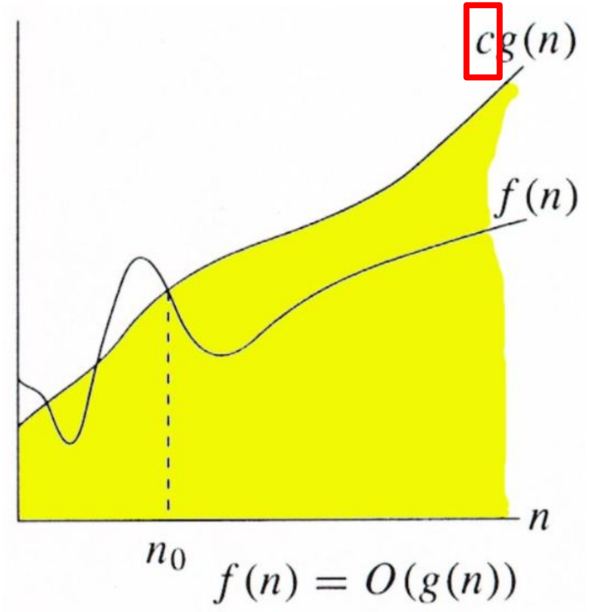


Big-O Formal Definition

f(n) = O(g(n)) if:

- there exist constant *c > 0*

This is like “tweaking the slope” of g(n)



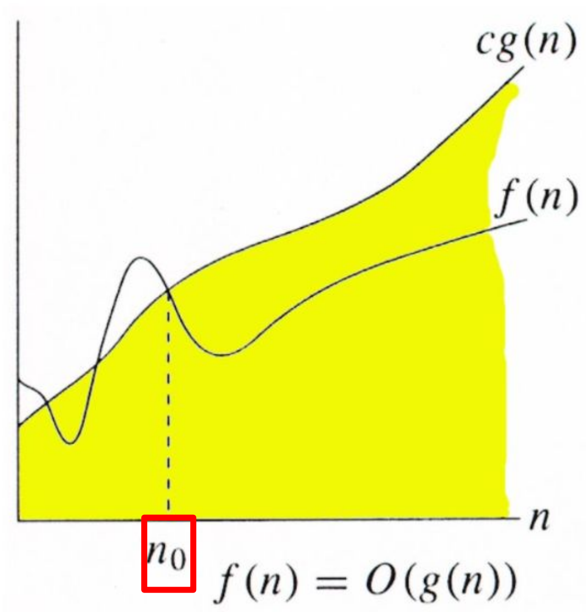
Big-O Formal Definition

f(n) = O(g(n)) if:

This is like "tweaking the slope" of g(n)

- there exist constant *c > 0*
- and there exist constant $n_0 > 0$

Like finding a start point



Big-O Formal Definition

f(n) = O(g(n)) if:

This is like "tweaking the slope" of g(n)

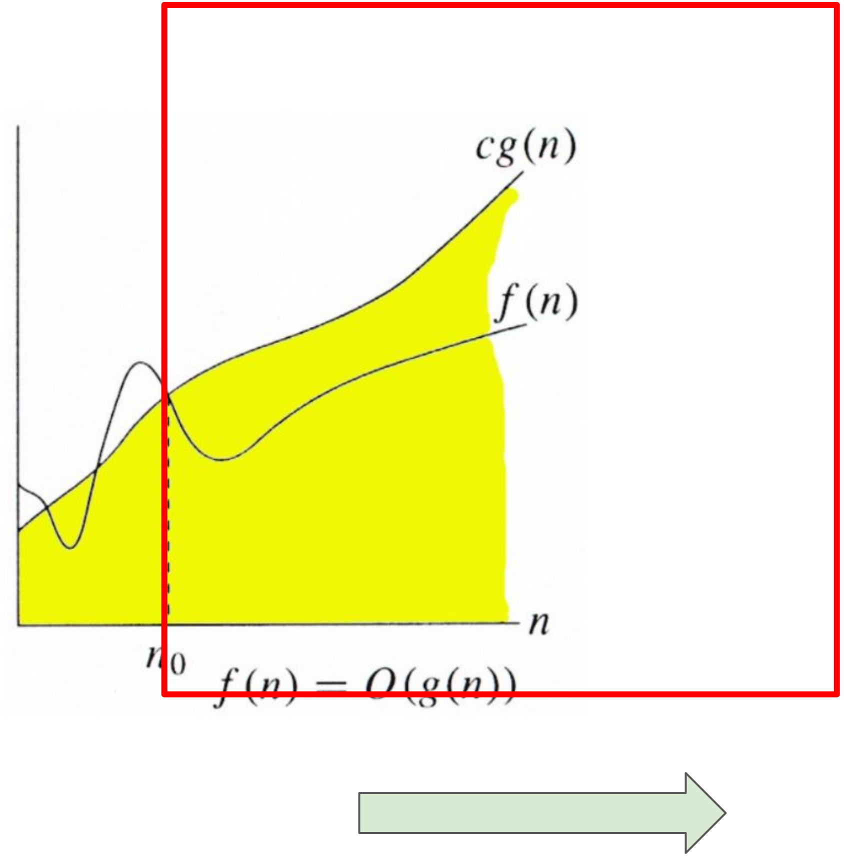
- there exist constant $c > 0$
- and there exist constant $n_0 > 0$

Like finding a start point

such that:

- $0 \le f(n) \le cg(n)$ for all $n \ge n_0$

After that start point. The function upper-bounding function "always stays above"



Little-o notation and little-omega notation

Analogy:

- O-notation vs o-notation is like $\leq$ vs $<$
- Ω-notation vs ω-notation is like $\geq$ vs $>$

Little-o formal definition, compared to Big-O

$f(n) = o(g(n))$ if:

$f(n) = O(g(n))$ if:

Little-o formal definition, compared to Big-O

$f(n) = o(g(n))$ if:

- for ALL constant $c > 0$

$f(n) = O(g(n))$ if:

- there exist constant $c > 0$

Little-o formal definition, compared to Big-O

$f(n) = o(g(n))$ if:

- for ALL constant $c > 0$
- and there exist constant $n_0 > 0$

$f(n) = O(g(n))$ if:

- there exist constant $c > 0$
- and there exist constant $n_0 > 0$

Little-o formal definition, compared to Big-O

$f(n) = o(g(n))$ if:

- for ALL constant $c > 0$
- and there exist constant $n_0 > 0$

such that:

- $0 \leq f(n) < cg(n)$ for all $n \geq n_0$

$f(n) = O(g(n))$ if:

- there exist constant $c > 0$
- and there exist constant $n_0 > 0$

such that:

- $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$

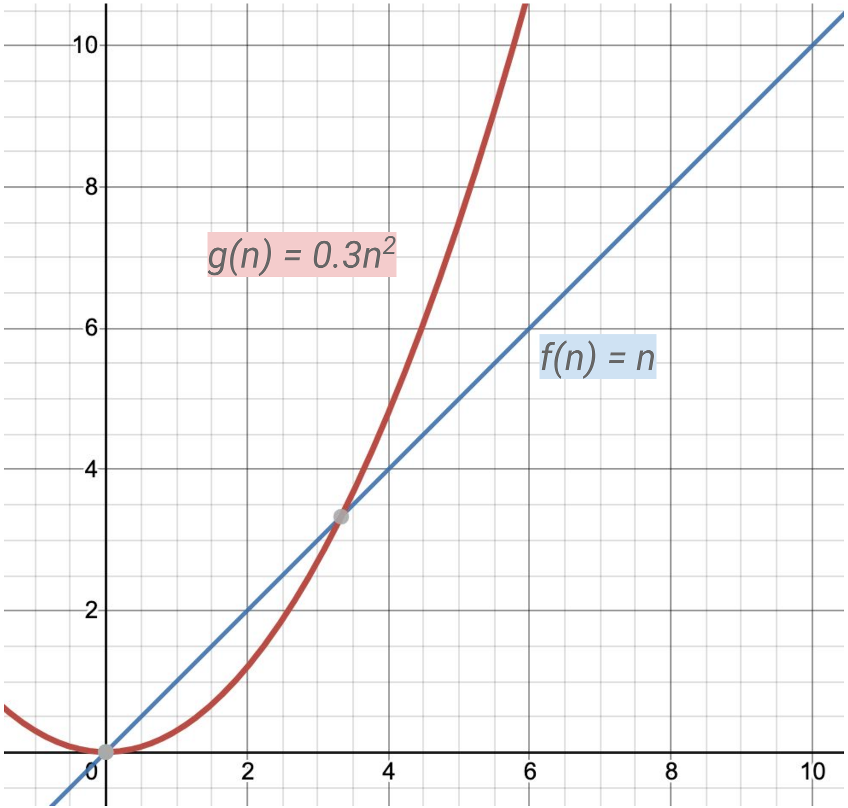
Little-o formal definition, compared to Big-O

$f(n) = o(g(n))$ if:

- for ALL constant $c > 0$
- and there exist constant $n_0 > 0$

such that:

- $0 \leq f(n) < cg(n)$ for all $n \geq n_0$



Little-o formal definition, compared to Big-O

$f(n) = o(g(n))$ if:

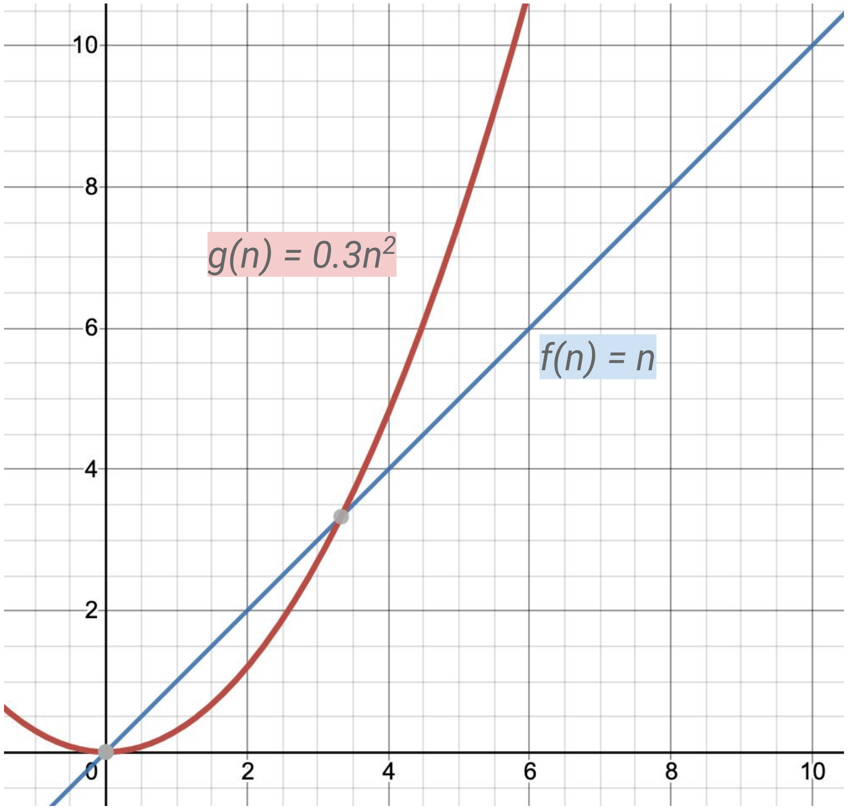
- for ALL constant $c > 0$
- and there exist constant $n_0 > 0$

such that:

- $0 \leq f(n) < cg(n)$ for all $n \geq n_0$

Intuition:

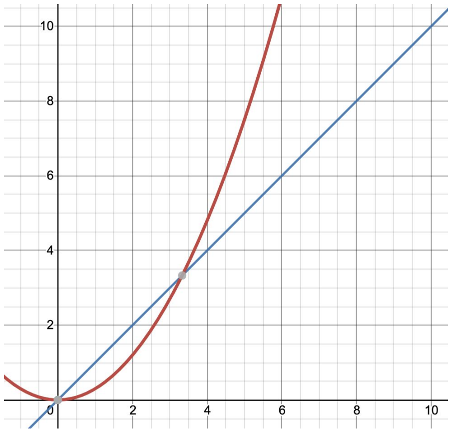
- No matter how you tweak the slope of g(n),
- At some point, g(n) will "overtake" f(n)



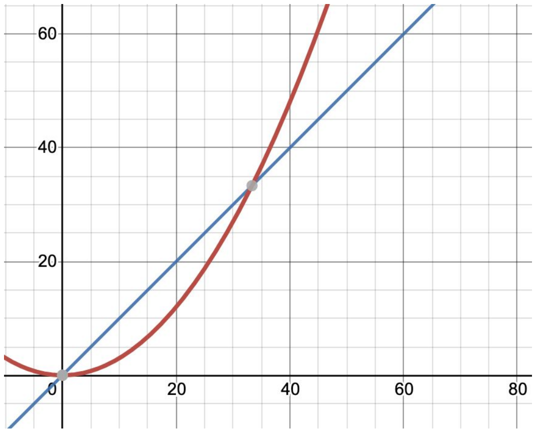
Little-o formal definition, compared to Big-O

Intuition:

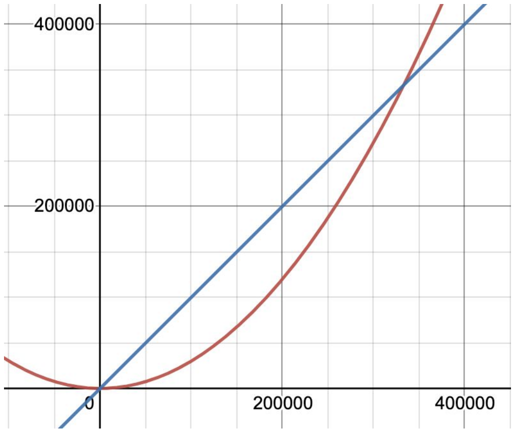
- No matter how you tweak the slope of g(n),
- At some point, g(n) will “overtake” f(n)



$0.3n^2$ vs n



$0.03n^2$ vs n



$0.000003n^2$ vs n

Disproving little-o (by definition)

$n^2 - n$ is not $o(n^2)$

$f(n) = o(g(n))$ if:

- for ALL constant $c > 0$
- and there exist constant $n_0 > 0$

such that:

- $0 \leq f(n) < cg(n)$ for all $n \geq n_0$

Disproving little-o (by definition)

$n^2 - n$ is not $o(n^2)$

Idea: Find a counterexample to the constant c, such that you cannot find any suitable $n_{.0}$

$f(n) = o(g(n))$ if:

- for ALL constant $c > 0$
- and there exist constant $n_0 > 0$

such that:

- $0 \le f(n) < cg(n)$ for all $n \ge n_0$

Disproving little-o (by definition)

$n^2 - n$ is not $o(n^2)$

Idea: Find a counterexample to the constant c, such that you cannot find any suitable $n_{.0}$

Intuition: You choose c. Then this c causes the “upper bounding graph” to ALWAYS (after a certain n_0) be **lower** than what is is trying to bound

$f(n) = o(g(n))$ if:

- for ALL constant $c > 0$
- and there exist constant $n_0 > 0$

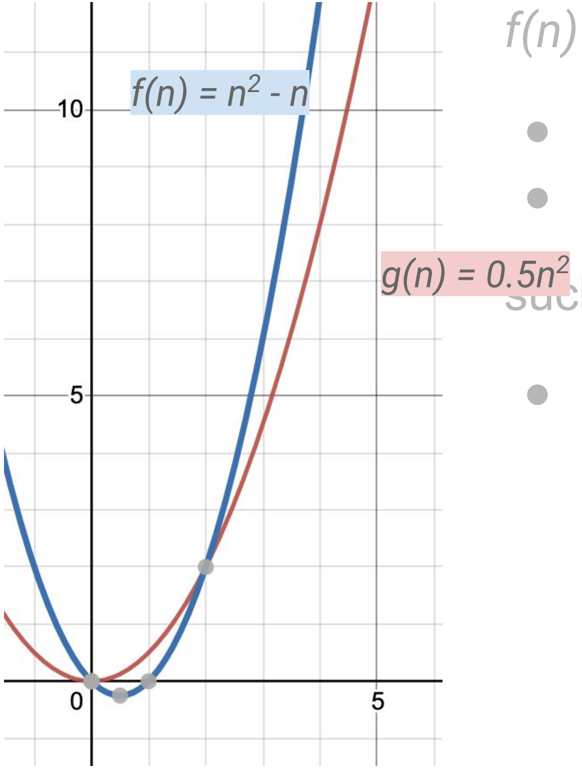
such that:

- $0 \leq f(n) < cg(n)$ for all $n \geq n_0$

Disproving little-o (by definition)

$n^2 - n$ is not $o(n^2)$

Example: c is 0.5



$f(n) = o(g(n))$ if:

- for ALL constant $c > 0$
- and there exist constant $n_0 > 0$

such that:

- $0 \le f(n) < cg(n)$ for all $n \ge n_0$

$n^2 - n \quad \text{vs} \quad 0.5n^2$

"After some n, n^2-n is always above $0.5n^2$"

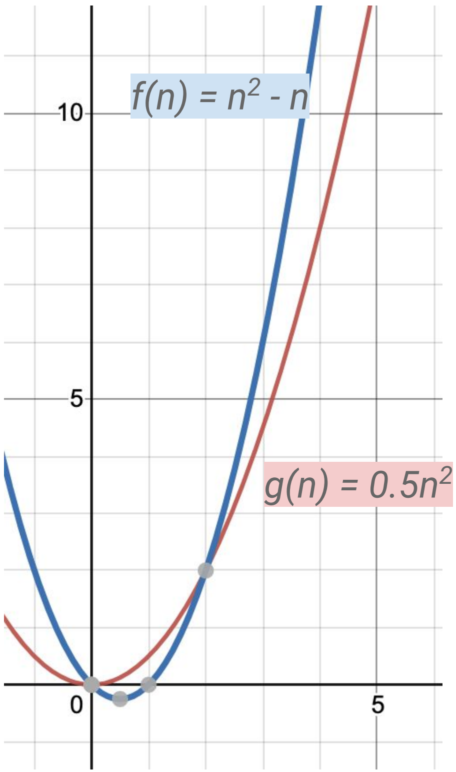
$$n^2 - n = 0.5n^2 + 0.5n^2 - n$$
$$\geq 0.5n^2 - n \qquad [0.5n^2 \geq 0 \ \forall n]$$

AND

$0.5n^2 - n \geq 0$ when $n \geq 2$

$\therefore$ For $n \geq 2$

$$n^2 - n = 0.5n^2 + 0.5n^2 - n$$
$$\geq 0.5n^2 \qquad [0.5n^2 - n \geq 0 \text{ when } n \geq 0]$$



Limit Versions

- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \Rightarrow f(n) = o(g(n))$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \Rightarrow f(n) = O(g(n))$

Limit Versions

Intuition: g(n) is so large, that f(n) is "insignificant"

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

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \lim_{n\to\infty}\left(\frac{n}{n^2}\right) = \lim_{n\to\infty}\left(\frac{1}{n}\right) = 0$$

- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \Rightarrow f(n) = o(g(n))$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \Rightarrow f(n) = O(g(n))$

Limit Versions

- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \Rightarrow f(n) = o(g(n))$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \Rightarrow f(n) = O(g(n))$

Intuition: If g(n) is so large, then f(n) is insignificant.
(This is the little-o case)

But what if they are “roughly equal”?

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

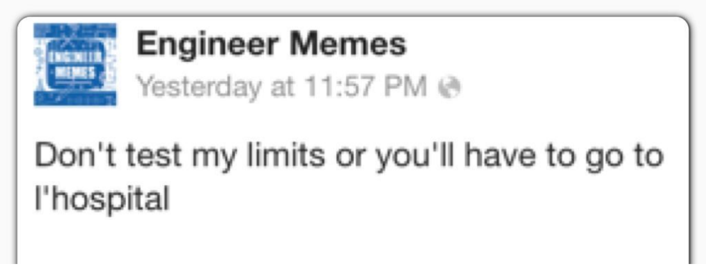
$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \lim_{n\to\infty}\left(\frac{2n^2}{n^2}\right) = \lim_{n\to\infty}(2) = 2$$

Limit Versions (theta and omega)

- $0 < \lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \Rightarrow f(n) = \Theta(g(n))$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) > 0 \Rightarrow f(n) = \Omega(g(n))$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty \Rightarrow f(n) = \omega(g(n))$

L'Hopital's Rule

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = \lim_{x \to \infty} \frac{f'(x)}{g'(x)}$$



L’Hopital’s Rule

Differentiation with respect to x

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \boxed{\frac{f'(x)}{g'(x)}}$$

Engineer Memes

Yesterday at 11:57 PM

Don't test my limits or you'll have to go to l'hospital

Example

$$\lim_{n\to\infty} \frac{n \log n}{n^2}$$

$$= \lim_{n\to\infty} \frac{\log n}{n}$$

$$= \lim_{n\to\infty} \frac{1/n}{1}$$

L'Hopital's rule

$$= \lim_{n\to\infty} \frac{1}{n} = 0$$

$$\implies n \log n \in o(n^2)$$

Useful fact on logarithm and polynomial

$lg(n) = o(n^k)$ for all $k > 0$

Useful fact on logarithm and polynomial

$lg(n) = o(n^k)$ *for all* $k > 0$

This even means that $lg(n) = o(n^{0.0000000000000000000000000000000000000000000000000001})$

Exercise: Prove it!

Question 3

Which of the following is true?

A. $f(n) = o(g(n))$
B. $f(n) = \Theta(g(n))$
C. $f(n) = \omega(g(n))$

when $f(n) = \ln(n)$ and $g(n) = \log_{10}(n)$.

Question 3 (Solution)

$\boxed{\text{when } f(n) = \ln(n) \text{ and } g(n) = \log_{10}(n).}$

Answer: B. $f(n) = \Theta(g(n))$

- $\log_{10} n = \ln(n) / \ln(10)$.
- So, $g(n) \le f(n) \le \ln(10) \cdot g(n)$

Intuition: The change of base makes it "about the same"

Question 3 (Solution)

$\boxed{\text{when } f(n) = \ln(n) \text{ and } g(n) = \log_{10}(n).}$

Answer: B. $f(n) = \Theta(g(n))$

- $\boxed{\log_{10} n = \ln(n)/\ln(10).}$
- So, $\underbrace{g(n) \le f(n)}_{\substack{\text{e.g. } \log_{10}(10) \le \log_e(10) \\ 1 \le 2.30...}} \boxed{\le \ln(10) \cdot g(n)}$

Intuition: The change of base makes it "about the same"

Question 4

Which of the following is true?

A. $f(n) = o(g(n))$
B. $f(n) = \Theta(g(n))$
C. $f(n) = \omega(g(n))$

Note: $\log^4 n = (\log n)^4$

when $f(n) = n^{2.5}$ and $g(n) = n^2 \log^4 n$.

Question 4 (Intuition)

$$\boxed{\text{when } f(n) = n^{2.5} \text{ and } g(n) = n^2 \log^4 n.}$$

$$f(n) = n^{2.5} = n^2 \cdot n^{0.1} \cdot n^{0.1} \cdot n^{0.1} \cdot n^{0.1} \cdot n^{0.1}$$

$$g(n) = n^2 \log^4 n = n^2 \cdot \log n \cdot \log n \cdot \log n \cdot \log n$$

Using *$lg(n) = o(n^k)$ for all $k > 0$*, we "upper bound" all the log(n) in g(n) by the $n^{0.1}$ in f(n)

Question 4 (Solution)

$$\boxed{\text{when } f(n) = n^{2.5} \text{ and } g(n) = n^2 \log^4 n.}$$

Answer: C. $f(n) = \omega(g(n))$

$$\frac{f(n)}{g(n)} = \frac{n^{0.5}}{\log^4 n}$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Why? Try using L'Hopital's rule repeatedly!

L'hopital Rule Idea

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \lim_{n\to\infty}\left(\frac{n^{0.5}}{\ln^4 n}\right)$$

Note: ln is used instead of lg just for simplicity of differentiation.

$$= \lim_{n\to\infty}\left(\frac{0.5n^{-0.5}}{\frac{4\ln^3 n}{n}}\right)$$

L'hopital Rule

$$= \lim_{n\to\infty}\left(\frac{n^{0.5}}{8\ln^3 n}\right)$$

Notice that the power in denominator decreases, while in numerator it stays the same

and repeat…

Question 5

Which of the following is true?

A. $f(n) = o(g(n))$
B. $f(n) = \Theta(g(n))$
C. $f(n) = \omega(g(n))$

when $f(n) = 3^n$ and $g(n) = 2^n$.

Question 5 (Intuition)

I generally think of trying to compare something I already know: 2^n vs 4^n

So $f(n) = 4^n$, $g(n) = 2^n$

Question 5 (Intuition)

I generally think of trying to compare something I already know: 2^n vs 4^n

So $f(n) = 4^n$, $g(n) = 2^n$

Notice that $f(n) = (2^{2n}) = (2^n)^2 = (g(n))^2$

Not hard to see that $f(n) = \omega(g(n))$ [think of $f(n) = n^2$ and $g(n) = n$]

Question 5 (Intuition)

I generally think of trying to compare something I already know: 2^n vs 4^n

So $f(n) = 4^n$, $g(n) = 2^n$

Notice that $f(n) = (2^{2n}) = (2^n)^2 = (g(n))^2$

Not hard to see that $f(n) = \omega(g(n))$ [think of $f(n) = n^2$ and $g(n) = n$]

So it gives a “guess” on what answer it will be. Remains to rigorously prove the 3^n vs 2^n version

Question 5 (Solution)

$\boxed{\text{when } f(n) = 3^n \text{ and } g(n) = 2^n.}$

Answer: C. $f(n) = \omega(g(n))$

$$\frac{f(n)}{g(n)} = \frac{3^n}{2^n} = 1.5^n$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Question 6

- Ali has **81 coconuts**, all of which have the **same weight, except for one** which is heavier
- He does not know which is the heavier coconut
- Ali's friend has a balance scale, but will **charge Ali one dollar** for each use of the scale
- What is the maximum amount of money that Ali has to pay to guarantee that he can find the heaviest coconut, assuming that Ali uses an algorithm?

Options: 3, 4, 5, 6

Question 6 (Intuition)

Let’s say we only have 3 coconuts. How do we know which one is the heavier coconut?

(smol)
coconut A

(smol)
coconut B

(beeg)
coconut
C

Question 6 (Intuition)

Compare two of them:

- If you found one heavier than the other, that is the heavier coconut!
- Otherwise, the last coconut is the heavier coconut

(smol)
coconut A

(smol)
coconut B

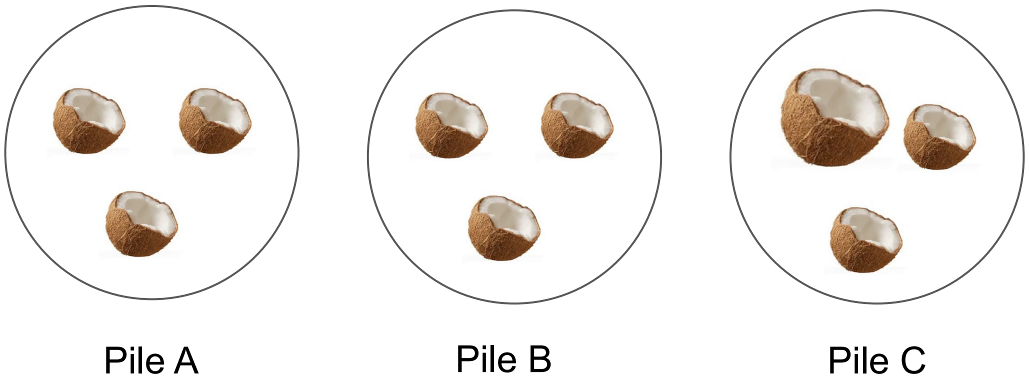
(beeg)
coconut
C

Question 6 (Intuition)

What if you **have 9 coconuts**?

Question 6 (Intuition)

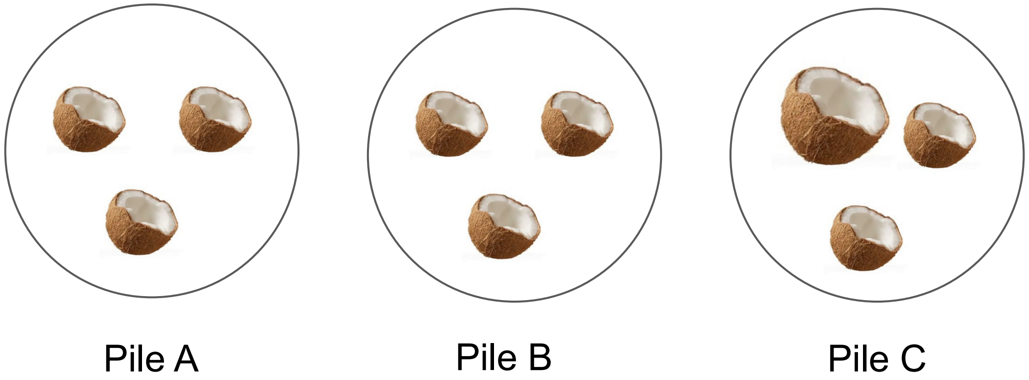
What if you **have 9 coconuts**? Divide into 3 piles! (Remember, they have the same weight except for one of them)



Question 6 (Intuition)

What if you **have 9 coconuts**? Divide into 3 piles! (Remember, they have the same weight except for one of them)

- Compare two coconut piles, and recurse on the one you know to be heavier!
- Eg the next step here is to recurse on the heavier pile with 3 coconuts



Question 6 (Solution)

- 3 coconuts → 1 weightings
- 9 coconuts → 2 weightings
- 27 coconuts → 3 weightings
- 81 coconuts → 4 weightings

Therefore, the **maximum cost is 4**!

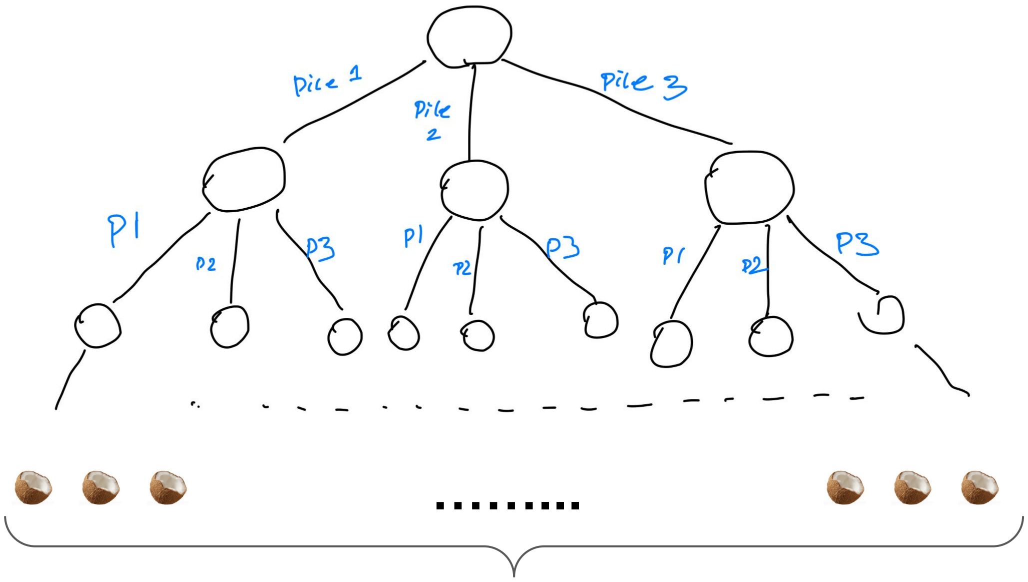
Question 6 (Note)

- To see that this is optimal, note that the scale can divide the coconuts into **at most 3 groups** with each weighting
- Any algorithm using only the scale can described as:
  - **Ternary** decision tree
  - **Heavy coconut at each leaf**

How many leaves are there if we have 81 coconuts?

Question 6 (Note)

How many leaves are there if we have 81 coconuts? **81 leaves as well!**



81 possible coconuts as the answer (heavy coconut)

Question 6 (Note)

We claim that 4 is really optimal - What if we **only allow 3 comparisons?**

Question 6 (Note)

We claim that 4 is really optimal - What if we **only allow 3 comparisons?**

Then our ternary tree will only have $3^3 = 27$ leaves. **Not enough to cover all 81 possibilities!**