Review Section 2/9

Big O notation

Main Idea

Focus on how the runtime **scales** with n (the input size).

Some examples…

(Only pay attention to the largest function of n that appears.)

Number of operations	Asymptotic Running Time
$\frac{1}{10}e^n + 10n^2$	$O(e^n)$
$n^3 + 2n^2 + 7$	$O(n^3)$
$0.1\sqrt{n} + 10^9 n^{0.05}$	$O(\sqrt{n})$
$11\log(n) + 1$	$O(\log(n))$

We say this algorithm is "asymptotically faster" than the others.

Informal definition for O(…)

- Let $T(n)$, $g(n)$ be functions of positive integers.
  - Think of $T(n)$ as a runtime: positive and increasing in n.

- We say “$T(n)$ is $O\left(g(n)\right)$” if:

for large enough n,

$T(n)$ is at most some constant multiple of $g(n)$.

“constant” means “some number that doesn’t depend on n.”

Formal definition of O(...)

- Let $T(n)$, $g(n)$ be functions of positive integers.
  - Think of $T(n)$ as a runtime: positive and increasing in n.
- Formally,

$$T(n) = O\big(g(n)\big)$$

"If and only if" "For all"

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$

$$T(n) \leq c \cdot g(n)$$

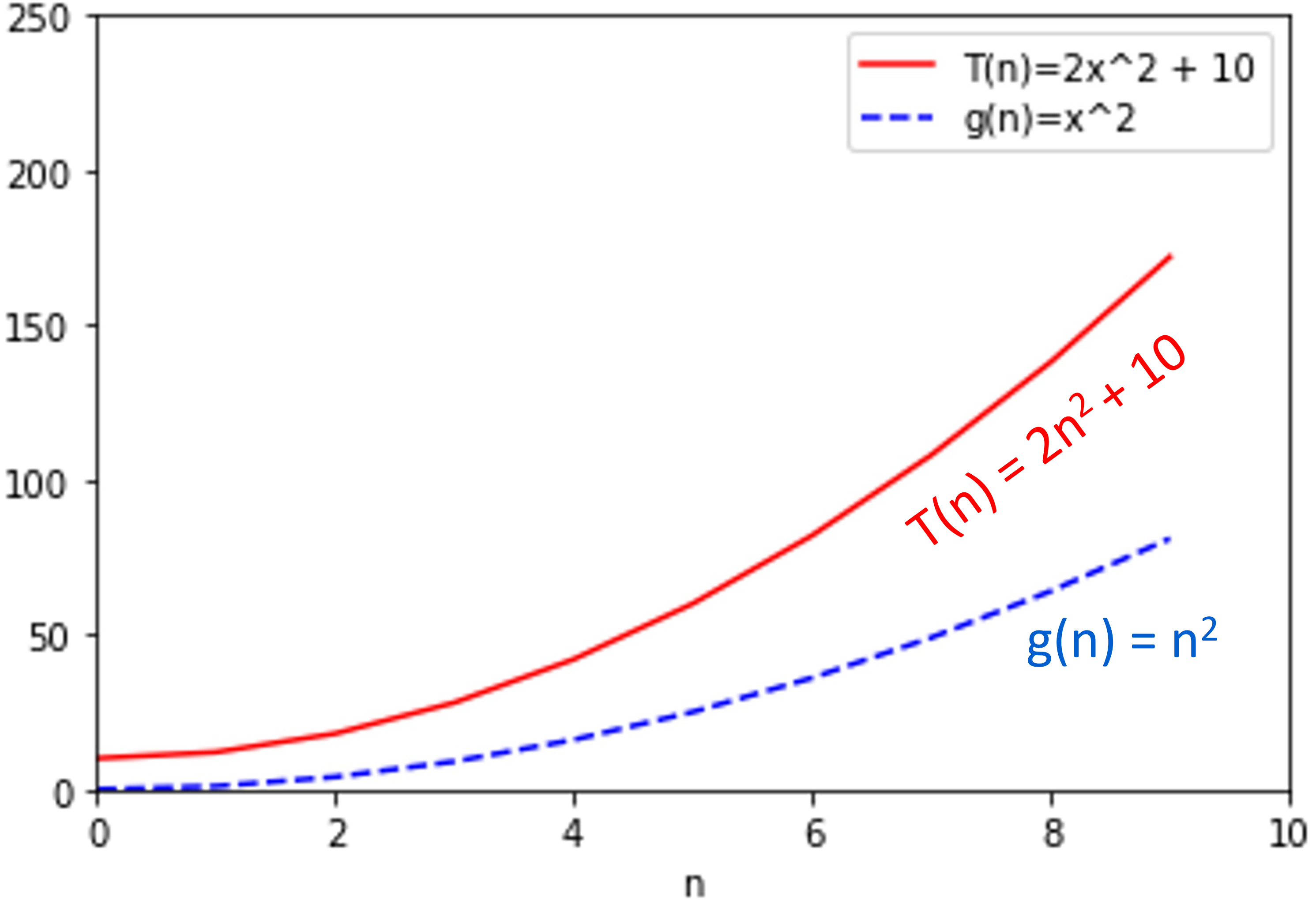
"There exists"

"such that"

Example

$2n^2 + 10 = O(n^2)$

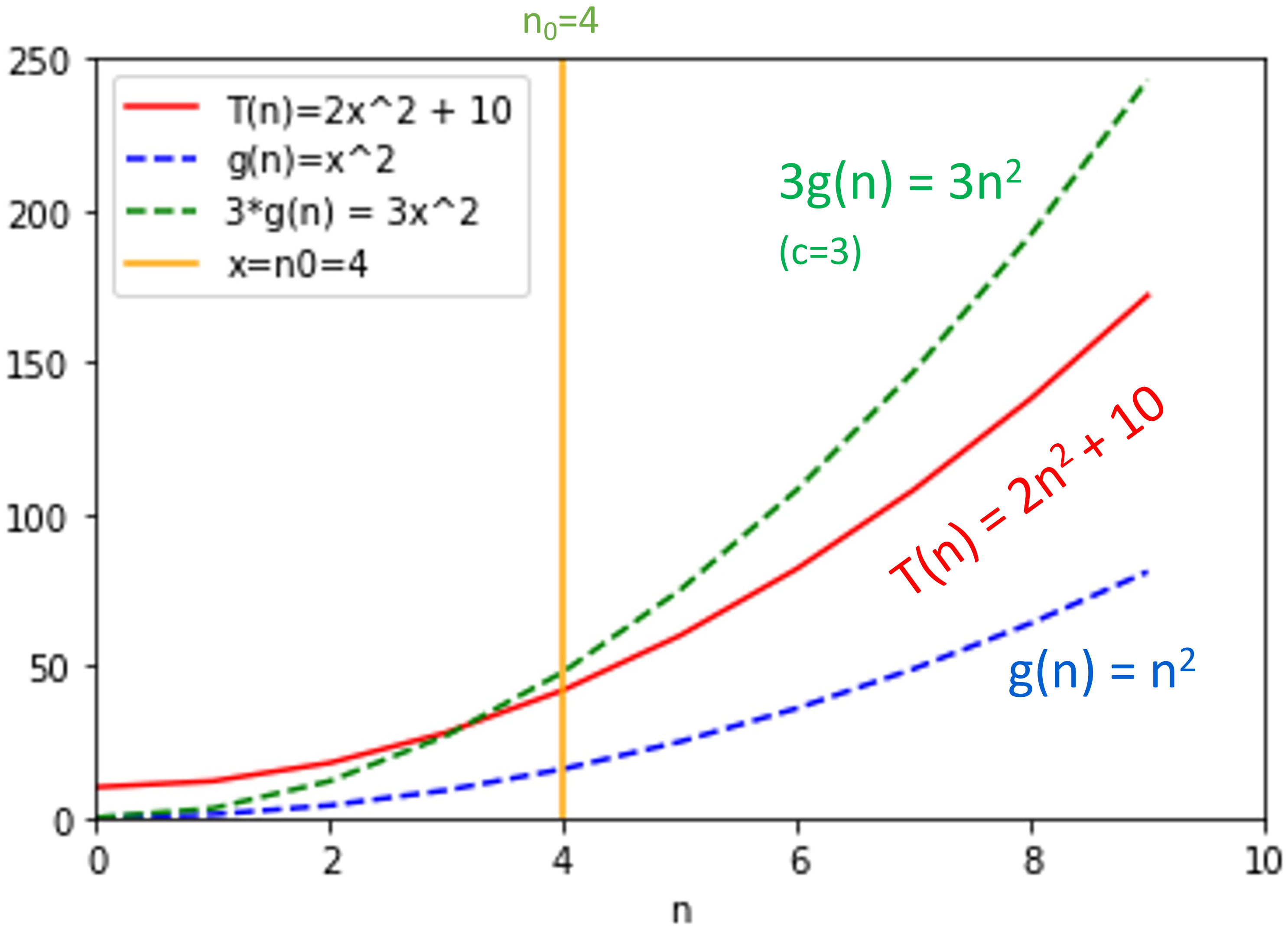
$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Example

$2n^2 + 10 = O(n^2)$

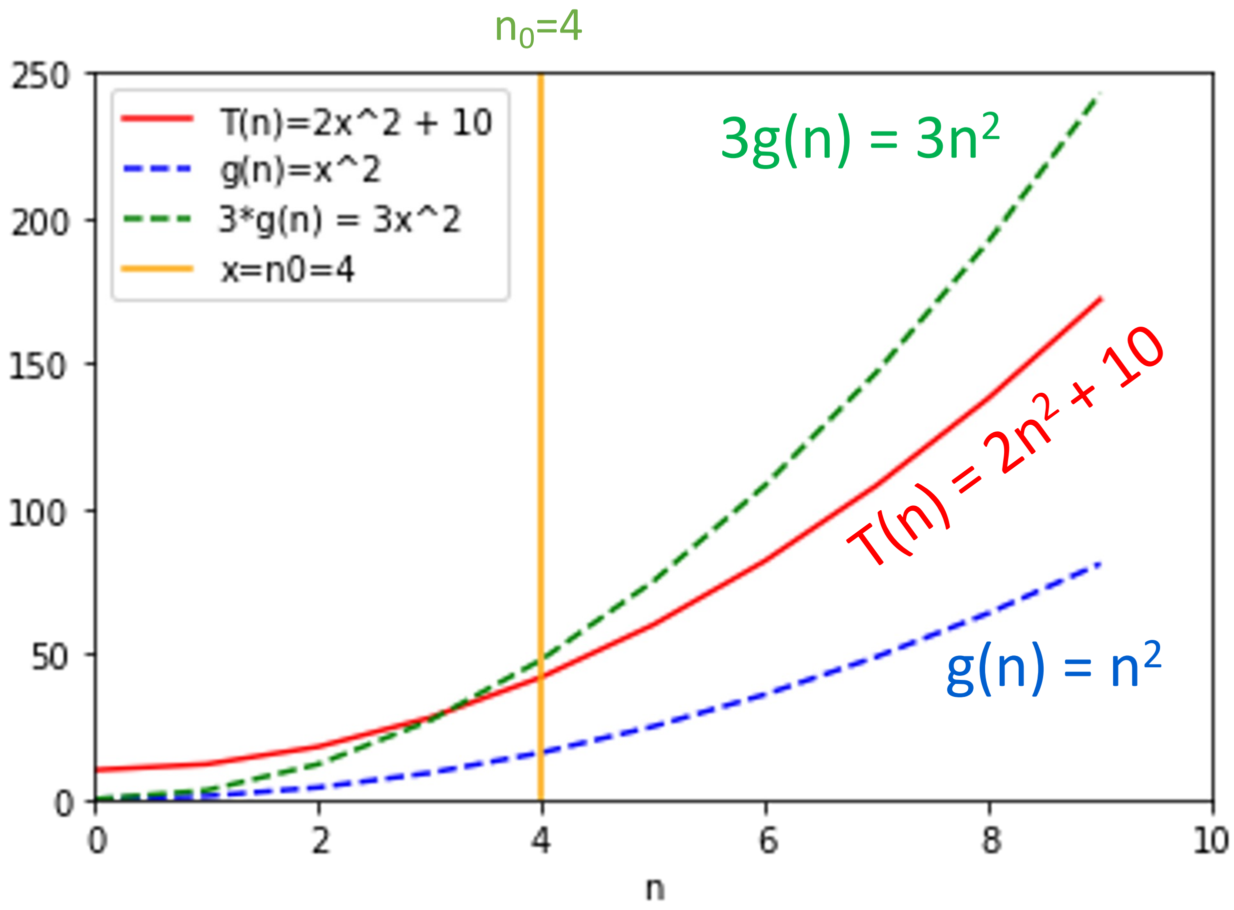
$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Example

$2n^2 + 10 = O(n^2)$

$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Formally:

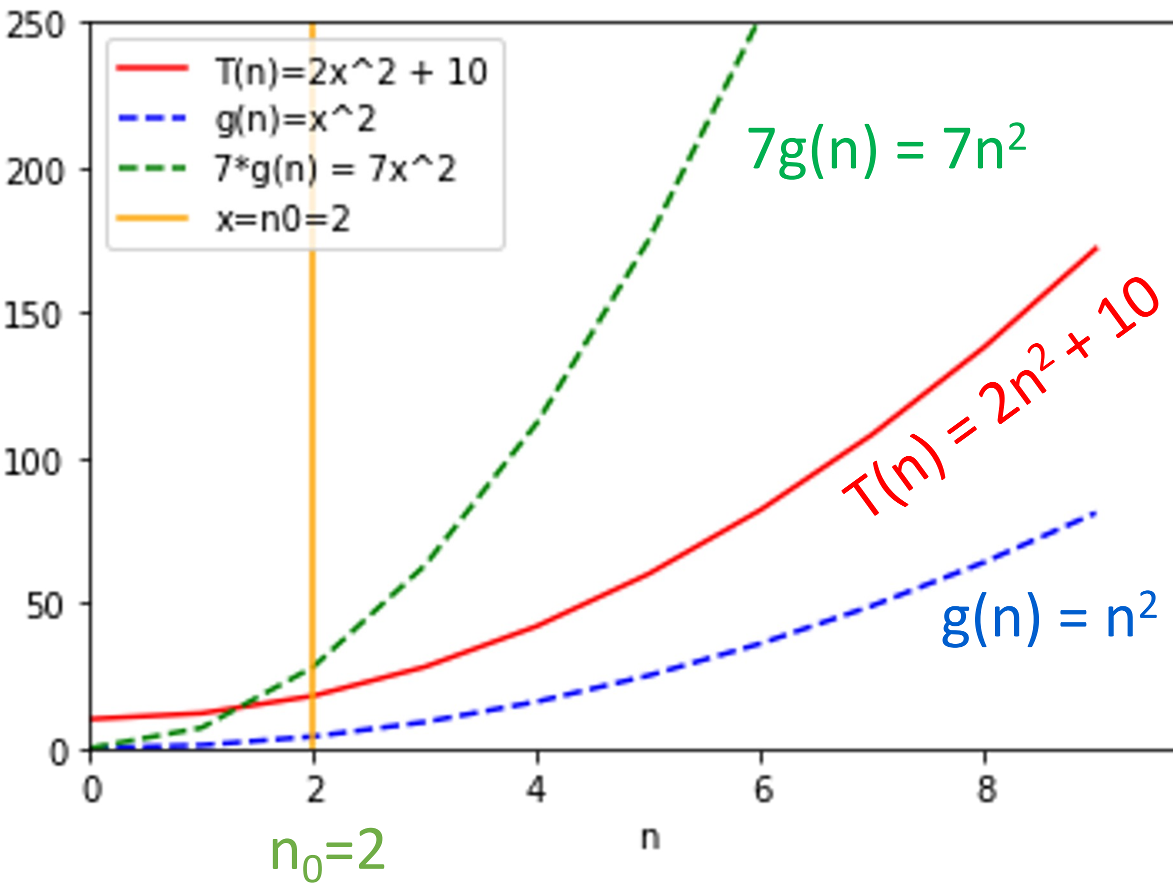
- Choose c = 3
- Choose n_0 = 4
- Then:

$$\forall n \geq 4,$$
$$2n^2 + 10 \leq 3 \cdot n^2$$

Same example

$2n^2 + 10 = O(n^2)$

$$T(n) = O(g(n)) \\ \Leftrightarrow \\ \exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0, \\ T(n) \leq c \cdot g(n)$$



Formally:

- Choose c = 7
- Choose $n_0 = 2$
- Then:

$$\forall n \geq 2,$$

$$2n^2 + 10 \leq 7 \cdot n^2$$

There is not a “correct” choice of c and n_0

Ω(…) means lower bound

- We say "$T(n)$ is $\Omega\big(g(n)\big)$" if, for large enough n, $T(n)$ is at least as big as a constant multiple of $g(n)$.

- Formally,

$$T(n) = \Omega\big(g(n)\big)$$

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$

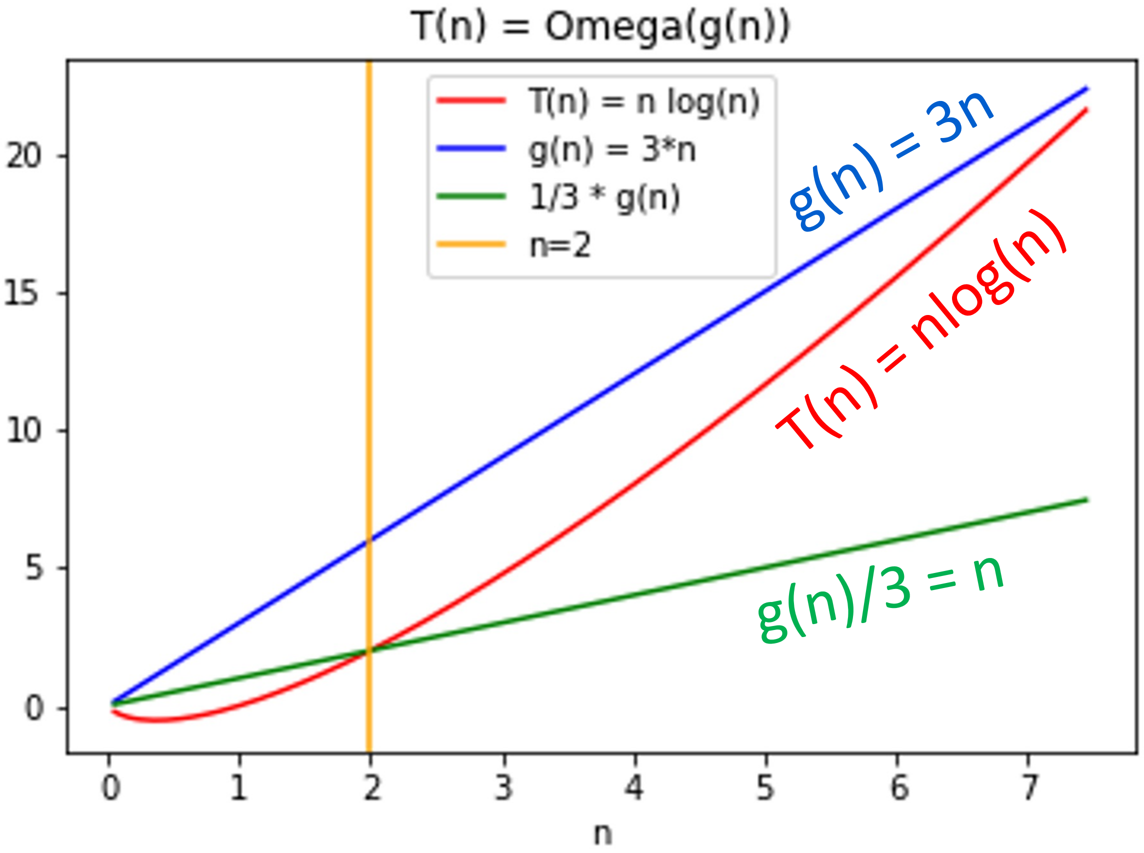
$$c \cdot g(n) \leq T(n)$$

Switched these!!

Example

$n \log_2(n) = \Omega(3n)$

$$T(n) = \Omega(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$c \cdot g(n) \leq T(n)$$



- Choose c = 1/3
- Choose n_0 = 2
- Then

$$\forall n \geq 2,$$

$$\frac{3n}{3} \leq n \log_2(n)$$

Θ(…) means both!

- We say "$T(n)$ is $\Theta(g(n))$" iff both:

$$T(n) = O\big(g(n)\big)$$

and

$$T(n) = \Omega\big(g(n)\big)$$

Induction

Background on Induction

- Type of mathematical proof
- Typically used to establish a given statement for all natural numbers (e.g. integers > 0)
- Proof is a sequence of deductive steps
    - Show the statement is true for the first number.
    - Show that if the statement is true for any one number, this implies the statement is true for the next number.
    - If so, we can infer that the statement is true for all numbers.

Components of Inductive Proof

Inductive proof is composed of 3 major parts :

- **Base Case** : One or more particular cases that represent the most basic case. (e.g. n=1 to prove a statement in the range of positive integer)
- **Induction Hypothesis** : Assumption that we would like to be based on. (e.g. Let's assume that P(k) holds)
- **Inductive Step** : Prove the next step based on the induction hypothesis. (i.e. Show that Induction hypothesis P(k) implies P(k+1))

Weak Induction vs Strong Induction:

- In weak induction, we only assume that particular statement holds at k-th step,
- In strong induction, we assume that the particular statement holds at all the steps from the base case to k-th step

Example: Integer Summation

Claim:

Let $S(n) = \sum_{i=1}^{n} i$. *Then* $S(n) = \frac{n(n+1)}{2}$.

Base Case:

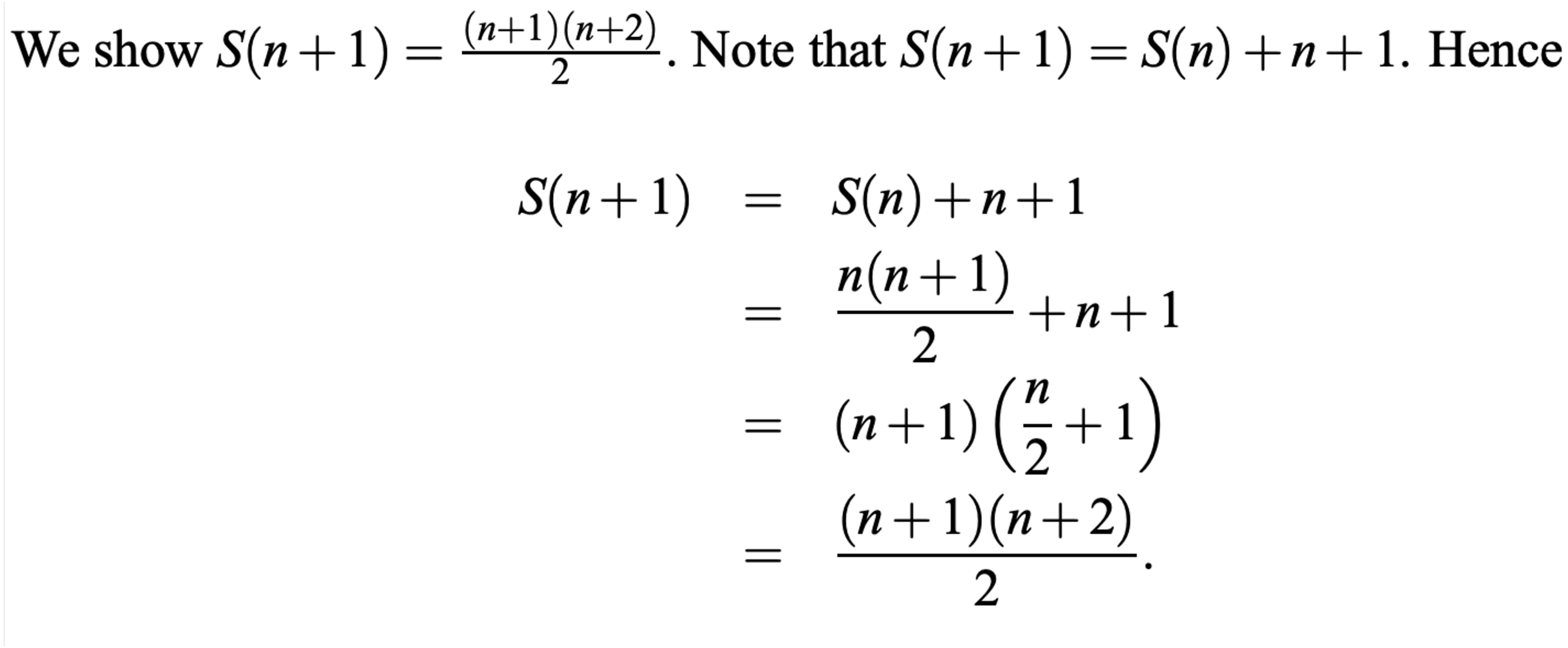
We show the statement is true for $n = 1$. As $S(1) = 1 = \frac{1(2)}{2}$, the statement holds.

Induction Hypothesis:

We assume $S(n) = \frac{n(n+1)}{2}$.

Example: Integer Summation

Inductive Step:



We show $S(n+1) = \frac{(n+1)(n+2)}{2}$. Note that $S(n+1) = S(n) + n + 1$. Hence

$$\begin{aligned} S(n+1) &= S(n) + n + 1 \\ &= \frac{n(n+1)}{2} + n + 1 \\ &= (n+1)\left(\frac{n}{2} + 1\right) \\ &= \frac{(n+1)(n+2)}{2}. \end{aligned}$$

Substitution Method

The Substitution Method

- Another way to solve recurrence relations.
- More general than the master method.

- Step 1: Generate a guess at the correct answer.
- Step 2: Try to prove that your guess is correct.
- (Step 3: Profit.)

First Example

- Consider the following problem:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \text{ with } T(0) = 0, T(1) = 1.$$

- The Master Method says $T(n) = O(n \log(n))$.
- We will prove this via the Substitution Method.

Step 1: Guess the answer

- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$

Expand $T\left(\frac{n}{2}\right)$

- $T(n) = 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$

Simplify

- $T(n) = 4 \cdot T\left(\frac{n}{4}\right) + 2n$

Expand $T\left(\frac{n}{4}\right)$

- $T(n) = 4 \cdot \left(2 \cdot T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n$

Simplify

- $T(n) = 8 \cdot T\left(\frac{n}{8}\right) + 3n$
- …

Guessing the pattern: $T(n) = 2^t \cdot T\left(\frac{n}{2^t}\right) + t \cdot n$

Plug in $t = \log(n)$, and get

$$T(n) = n \cdot T(1) + \log(n) \cdot n = n(\log(n) + 1)$$

You can guess the answer however you want: meta-reasoning, a little bird told you, wishful thinking, etc. One useful way is to try to “unroll” the recursion, like we’re doing here.

Step 2: Prove the guess is correct.

- Inductive Hypothesis: $T(n) = n(\log(n) + 1)$.
- Base Case (n=1): $T(1) = 1 = 1 \cdot (\log(1) + 1)$
- Inductive Step:
  - Assume Inductive Hyp. for $1 \leq n < k$:
    - Suppose that $T(n) = n(\log(n) + 1)$ for all $1 \leq n < k.$
  - Prove Inductive Hyp. for n=k:
    - $T(k) = 2 \cdot T\left(\frac{k}{2}\right) + k$ by definition
    - $T(k) = 2 \cdot \left(\frac{k}{2}\left(\log\left(\frac{k}{2}\right) + 1\right)\right) + k$ by induction.
    - $T(k) = k(\log(k) + 1)$ by simplifying.
    - So Inductive Hyp. holds for n=k.
- Conclusion: For all $n \geq 1, T(n) = n(\log(n) + 1)$

We're being sloppy here about floors and ceilings…what would you need to do to be less sloppy?

Step 3: Profit

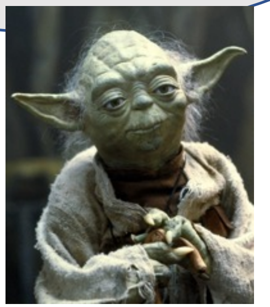
- Pretend like you never did Step 1, and just write down:

- **Theorem:** $T(n) = O(n \log(n))$
- **Proof: [Whatever you wrote in Step 2]**

Second Example

- $T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$ for $n > 10$.
- Base case: $T(n) = 1 \text{ when } 1 \leq n \leq 10$

Apply here, the Master Theorem does NOT.



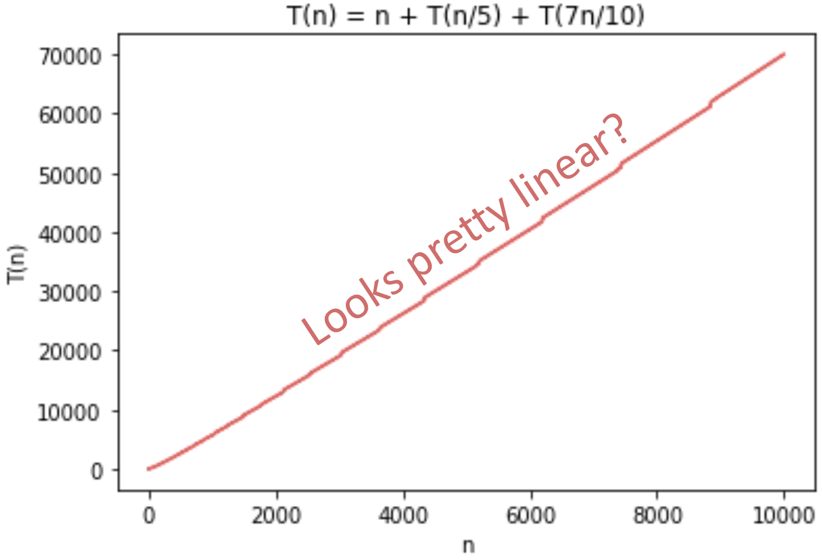
Jedi master Yoda

Step 1: guess the answer

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n \text{ for } n > 10.$$

Base case: $T(n) = 1$ when $1 \leq n \leq 10$

- Trying to work backwards gets gross fast…
- We can also just try it out.
- Let's guess $O(n)$ and try to prove it.



$T(n) \le T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$ for $n > 10$.

Base case: $T(n) = 1$ when $1 \le n \le 10$

Step 2: prove our guess is right

- Inductive Hypothesis: $T(n) \le \boldsymbol{C}n$
- Base case: $1 = T(n) \le \boldsymbol{C}n$ for all $1 \le n \le 10$
- Inductive step:
  - Let k > 10. Assume that the IH holds for all n so that $1 \le n < k$.
  - $$\begin{aligned} T(k) &\le k + T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right) \\ &\le k + \boldsymbol{C}\cdot\left(\frac{k}{5}\right) + \boldsymbol{C}\cdot\left(\frac{7k}{10}\right) \\ &= k + \frac{\boldsymbol{C}}{5}k + \frac{7\boldsymbol{C}}{10}k \\ &\le \boldsymbol{C}k\ ?? \end{aligned}$$
  - (aka, want to show that IH holds for n=k).
- Conclusion:
  - There is some $\boldsymbol{C}$ so that for all $n \ge 1$, $T(n) \le \boldsymbol{C}n$
  - By the definition of big-Oh, T(n) = O(n).

We don't know what C should be yet! Let's go through the proof leaving it as "C" and then figure out what works…

Whatever we choose C to be, it should have C≥1

Let's solve for C and make this true!
C = 10 works.

$$T(n) \leq n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) \text{ for } n > 10.$$

Base case: $T(n) = 1$ when $1 \leq n \leq 10$

Step 3: profit

Theorem: $T(n) = O(n)$

Proof:

- Inductive Hypothesis: $T(n) \leq \mathbf{10}n$.
- Base case: $1 = T(n) \leq \mathbf{10}n$ for all $1 \leq \mathrm{n} \leq 10$
- Inductive step:
  - Let k > 10. Assume that the IH holds for all n so that $1 \leq n < k$.
  - $$\begin{aligned} T(k) &\leq k + T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right) \\ &\leq k + \mathbf{10} \cdot \left(\frac{k}{5}\right) + \mathbf{10} \cdot \left(\frac{7k}{10}\right) \\ &= k + 2k + 7k = \mathbf{10}k \end{aligned}$$
  - Thus, IH holds for n=k.
- Conclusion:
  - For all $n \geq 1, T(n) \leq \mathbf{10}n$
  - Then, T(n) = O(n), using the definition of big-Oh with $n_0 = 1, c = 10$.

Linear Time Selection

The k select problem

- A is an array of size n, k is in {1,…,n}
- SELECT(A, k):
  - Return the k-th smallest element of A.

7	4	3	8	1	5	9	14

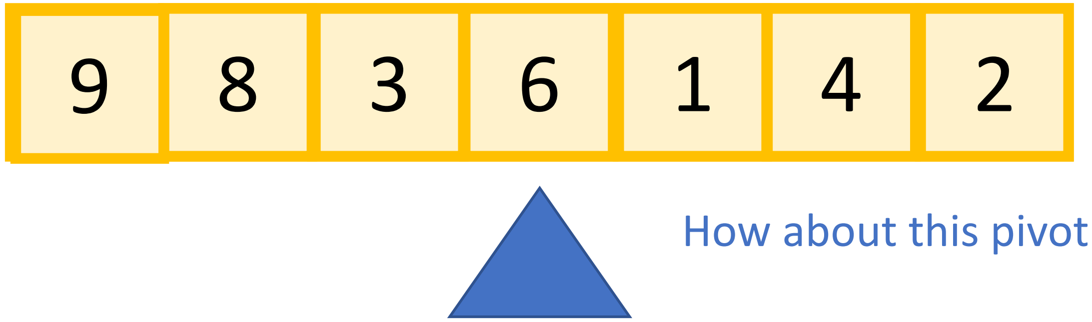
- SELECT(A, 1) = 1
- SELECT(A, 2) = 3
- SELECT(A, 3) = 4
- SELECT(A, 8) = 14

- SELECT(A, 1) = MIN(A)
- SELECT(A, n/2) = MEDIAN(A)
- SELECT(A, n) = MAX(A)

Being sloppy about floors and ceilings!

Idea: divide and conquer!

Say we want to find SELECT(A, k)



9	8	3	6	1	4	2

How about this pivot?

First, pick a “pivot.”
We’ll see how to do this later.

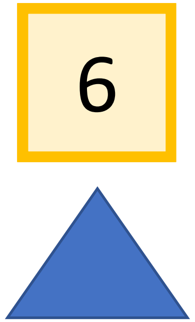
Next, partition the array into “bigger than 6” or “less than 6”

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Idea: divide and conquer!

Say we want to find SELECT(A, k)

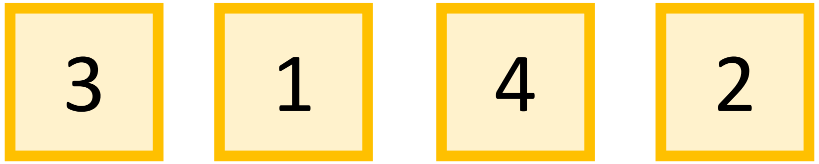


How about this pivot?

First, pick a "pivot."
We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"

This PARTITION step takes time O(n).
(Notice that we don't sort each half).

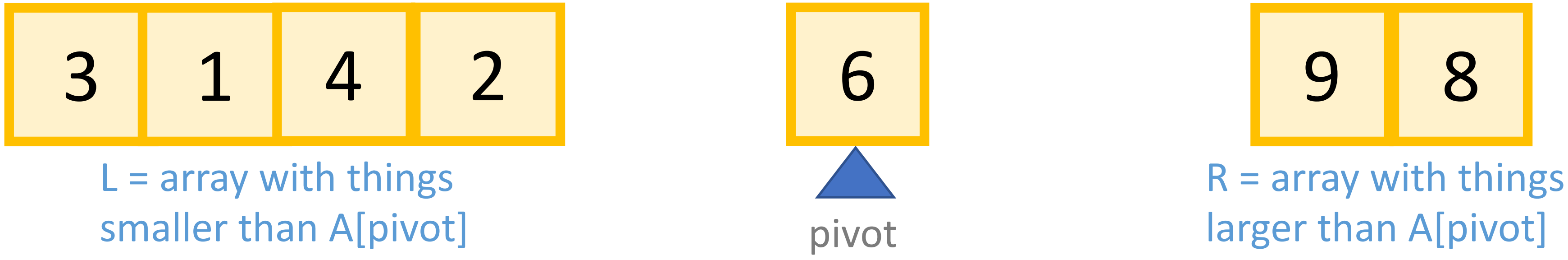


L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Idea continued…

Say we want to find SELECT(A, k)



L = array with things smaller than A[pivot]

pivot

R = array with things larger than A[pivot]

- If k = 5 = len(L) + 1:
  - We should return A[pivot]
- If k < 5:
  - We should return SELECT(L, k)
- If k > 5:
  - We should return SELECT(R, k – 5)

This suggests a recursive algorithm

(still need to figure out how to pick the pivot…)

Pseudocode

- `getPivot(A)` returns some pivot for us.
  - How?? We'll see later…
- `Partition(A,p)` splits up A into L, A[p], R.

```
Select(A,k):
    If len(A) <= 50:
        A = MergeSort(A)
        Return A[k-1]
    p = getPivot(A)
    L, pivotVal, R = Partition(A,p)
    if len(L) == k-1:
        return pivotVal
    Else if len(L) > k-1:
        return Select(L, k)
    Else if len(L) < k-1:
        return Select(R, k – len(L) – 1)
```

Base Case: If len(A) = O(1), then any sorting algorithm runs in time O(1).

Case 1: We got lucky and found exactly the k'th smallest value!

Case 2: The k'th smallest value is in the first part of the list

Case 3: The k'th smallest value is in the second part of the list

What is the running time?

- $$T(n) = \begin{cases} T(\mathbf{len}(\mathbf{L})) + O(n) & \mathbf{len}(\mathbf{L}) > k - 1 \\ T(\mathbf{len}(\mathbf{R})) + O(n) & \mathbf{len}(\mathbf{L}) < k - 1 \\ O(n) & \mathbf{len}(\mathbf{L}) = k - 1 \end{cases}$$

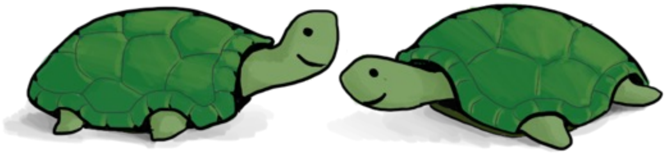
- What are **len(L)** and **len(R)**?
- That depends on how we pick the pivot…

The best way would be to always pick the pivot so that len(L) = k-1.
But say we don't have control over k, just over how we pick the pivot.

The ideal pivot

- We split the input exactly in half:
  - len(L) = len(R) = (n-1)/2

What happens in that case?



In case it's helpful…

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

$$T(n) = \begin{cases} O\left(n^d \log(n)\right) & \text{if } a = b^d \\ O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}$$

The idea pivot

- We split the input exactly in half:
  - len(L) = len(R) = (n-1)/2

- Let's pretend that's the case and use the **Master Theorem**!

- $T(n) \leq T\left(\frac{n}{2}\right) + O(n)$
- So a = 1, b = 2, d = 1
- $T(n) \leq O\left(n^d\right) = O(n)$

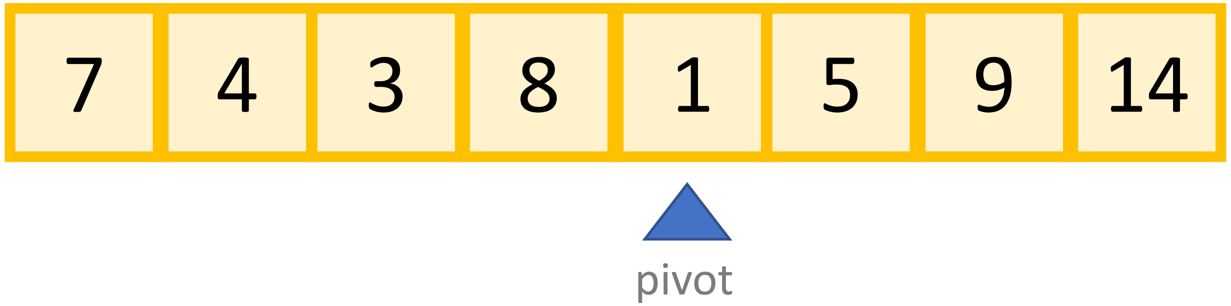
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

$$T(n) = \begin{cases} O\left(n^d \log(n)\right) & \text{if } a = b^d \\ O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}$$

That would be great!

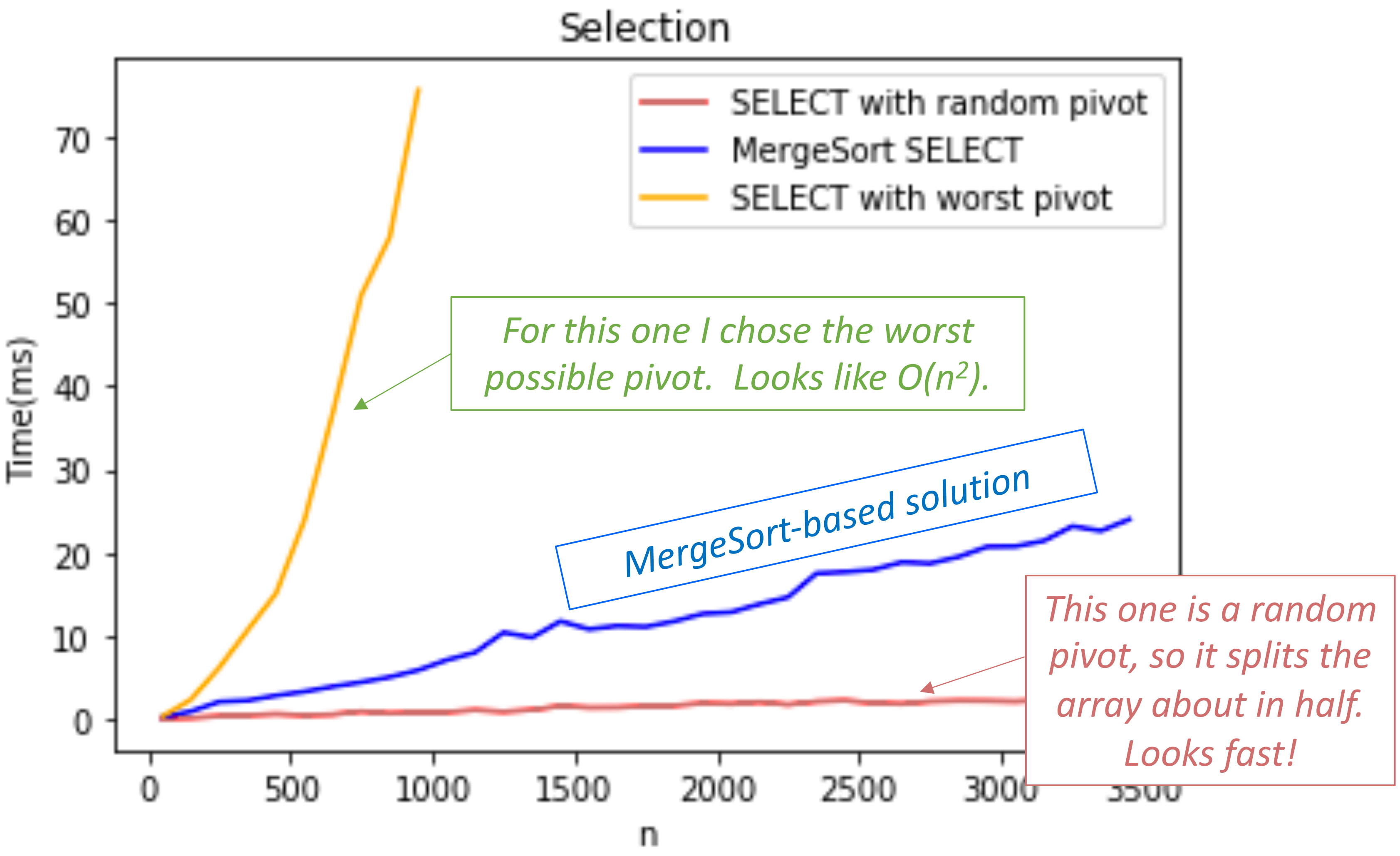
The worst pivot

- Say our choice of pivot doesn’t depend on A.
- A bad guy who **knows what pivots we will choose** gets to come up with A.



7	4	3	8	1	5	9	14

The distinction matters!



For this one I chose the worst possible pivot. Looks like $O(n^2)$.

MergeSort-based solution

This one is a random pivot, so it splits the array about in half. Looks fast!

See Lecture 4 Python notebook for code that generated this picture.

How do we pick our ideal pivot?

- We'd like to live in the ideal world.

- Pick the pivot to divide the input in half.
- Aka, pick the median!
- Aka, pick SELECT(A, n/2)!

How about a good enough pivot?

- We'd like to approximate the ideal world.

- Pick the pivot to divide the input about in half!
- Maybe this is easier!

A good enough pivot

- We split the input not quite in half:
  - 3n/10 < len(L) < 7n/10
  - 3n/10 < len(R) < 7n/10
- If we could do that (let's say, in time O(n)), the **Master Theorem** would say:
  - $T(n) \leq T\left(\frac{7n}{10}\right) + O(n)$
  - So a = 1, b = 10/7, d = 1
  - $T(n) \leq O\left(n^d\right) = O(n)$

STILL GOOD!

We still don't know that we can get such a pivot, but at least it gives us a goal and a direction to pursue!

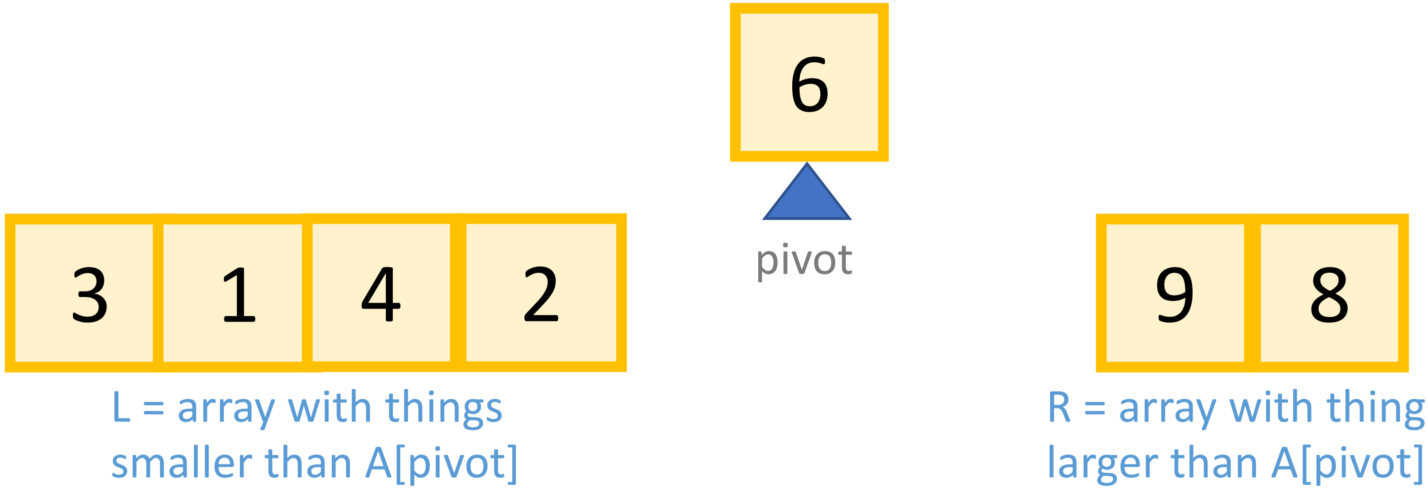
Lucky the lackadaisical lemur

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Goal

- Efficiently pick the pivot so that



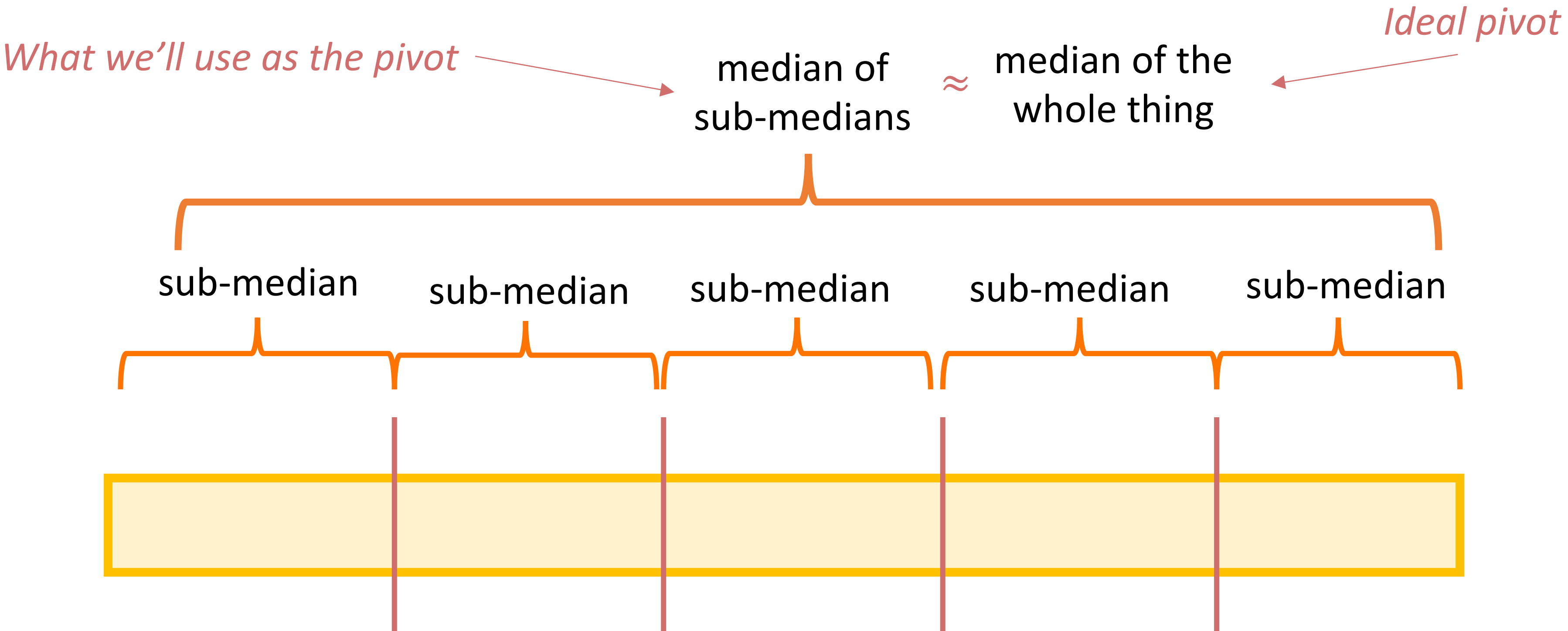
L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

$$\frac{3n}{10} < \text{len}(L) < \frac{7n}{10} \qquad \frac{3n}{10} < \text{len}(R) < \frac{7n}{10}$$

Another divide-and-conquer alg!

- We can't solve SELECT(A,n/2) (yet)
- But we can divide and conquer and solve SELECT(B,m/2) for smaller values of m (where len(B) = m).
- **Lemma*:** The median of sub-medians is close to the median.



What we'll use as the pivot

median of sub-medians ≈ median of the whole thing

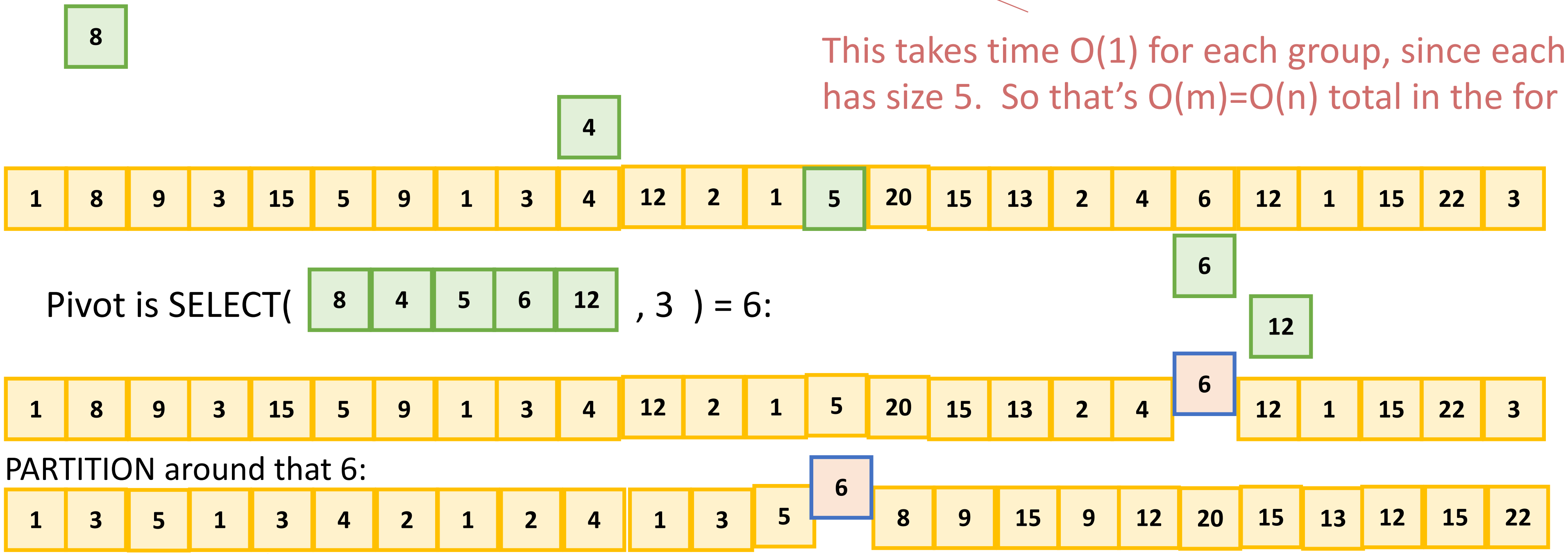
Ideal pivot

*we will make this a bit more precise.

How to pick the pivot

- CHOOSEPIVOT(A):
  - Split A into m = $\left\lceil \frac{n}{5} \right\rceil$ groups, of size <=5 each.
  - **For** i=1, .., m:
    - Find the median within the i'th group, call it p_i
  - p = SELECT($[\ p_1, p_2, p_3, \ldots, p_m\ ]$, m/2)
  - **return** the index of p in A

This takes time O(1) for each group, since each group has size 5. So that's O(m)=O(n) total in the for loop.



Pivot is SELECT(8 4 5 6 12 , 3) = 6:

PARTITION around that 6:

This part is L

This part is R: it's almost the same size as L.

This divides the array *approximately* in half

- Formally, we have:

Lemma: If we choose the pivots like this, then

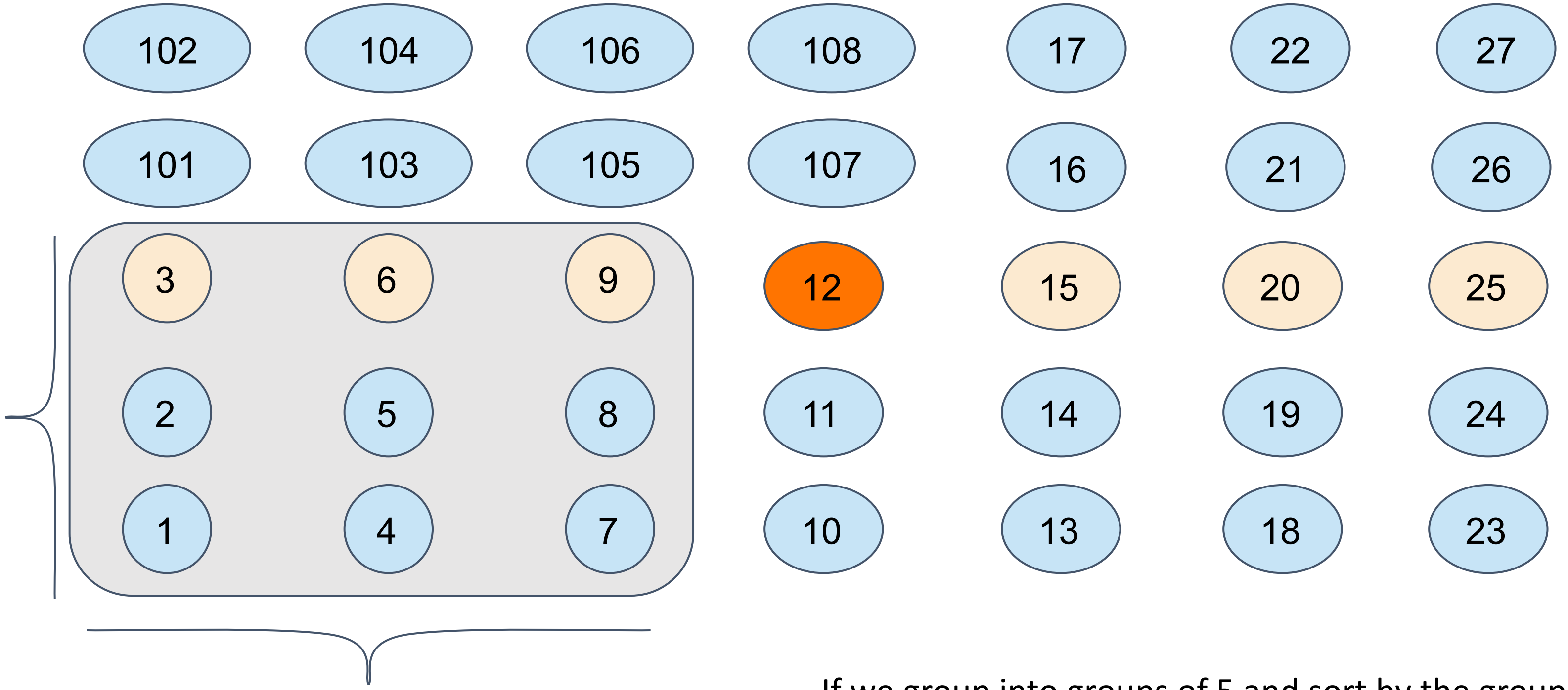
$$|L| \leq \frac{7n}{10} + 5$$

and

$$|R| \leq \frac{7n}{10} + 5$$

Why 70%/30% split worst case?

The most lopsided split that can happen after partitioning around the median of medians is 70/30.



At least ⅗ of each group is <= the group’s median.

At least ½ of the groups have medians <= pivot

If we group into groups of 5 and sort by the groups’ medians, the gray stuff (at least ⅗ * ½ = 3/10) all lies in one side of the partition.

How about the running time?

- Suppose the Lemma is true. (It is).
  - $|L| \leq \frac{7n}{10} + 5$ and $|R| \leq \frac{7n}{10} + 5$
- Recurrence relation:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

The call to CHOOSEPIVOT makes one further recursive call to SELECT on an array of size n/5.

Outside of CHOOSEPIVOT, there's at most one recursive call to SELECT on array of size 7n/10 + 5.

We're going to drop the "+5" for convenience,
but it does not change the final answer. Why?
Hint: Define T'(n) := T(n+1000) and write recurrence for T'

Siggi the Studious Stork

This sounds like a job for…

The Substitution Method!

Step 1: generate a guess
Step 2: try to prove that your guess is correct
Step 3: profit

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

That's convenient! We did this at the beginning of lecture!

Conclusion: $T(n) = O(n)$

Technically we only did it for $T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$, not when the last term has a big-Oh…

Plucky the Pedantic Penguin

Practice example

- Input:
  - Array A containing n possibly very large integers
  - k ranks r_0, ..., r_k, which are integers in the range {1, ..., n}
- Output:
  - Array B which contains the r_j-th smallest of the n integers, for every j in 1,...,k
- Requirement:
  - An $O(n \log k)$ algorithm

Practice example

- Find the median rank r_m using the Select algorithm
- Run Select algorithm to find a_m, the r_m-th smallest integer in A
- Recurse separately on
  - (i) the ranks and integers greater than r_m and a_m (respectively);
  - (ii) the ones smaller than r_m and a_m

- Runtime:
  - The recursion tree has a depth of $\log(k)$
  - At each level, the time spent if $O(n+k) = O(n)$
  - So in total $O(n \log k)$

Practice example

- We have an array of positive numbers $h_1, h_2, \cdots, h_n$
- The sum is $\sum_i h_i = C$
- The weighted median is defined as k such that:
    - $\sum_{i:\, h_i < h_k} h_i \leq \frac{C}{2}$
    - $\sum_{i:\, h_i > h_k} h_i \leq \frac{C}{2}$
- Goal: compute the weighted median in $O(n)$ worst case time

Practice example

- Find median h_k from $h_1, h_2, \cdots, h_n$
- Compute the sum of each side:
  - $H_L = \sum_{i:\, h_i < h_k} h_i$
  - $H_R = \sum_{i:\, h_i > h_k} h_i$
- If $H_L \le \frac{C}{2}$ and $H_R \le \frac{C}{2}$, return
- If $H_L > \frac{C}{2}$:
  - Change h_k to $h_k + H_R$, recurse on the elements smaller than h_k
- Else:
  - Change h_k to $h_k + H_L$, recurse on the elements larger than h_k

Quicksort

Quicksort

We want to sort this array.

7	6	3	5	1	2	4

First, pick a "pivot."

Do it at random.

random pivot!

This PARTITION step takes time O(n). (Notice that we don't sort each half). [same as in SELECT]

Next, partition the array into "bigger than 5" or "less than 5"

Arrange them like so:

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Recurse on L and R:

1	2	3	4	5	6	7

PseudoPseudoCode for what we just saw

```
QuickSort(A):
    If len(A) <= 1:
        return
    Pick some x = A[i] at random.  Call this the pivot.
    PARTITION the rest of A into:
        L (less than x) and
        R (greater than x)
    Replace A with  [L, x, R]  (that is, rearrange A in this order)
    QuickSort(L)
    QuickSort(R)
```

Running time?

- $T(n) = T(|L|) + T(|R|) + O(n)$
- In an ideal world…
  - if the pivot splits the array exactly in half…

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

- We’ve seen that a bunch:

$$T(n) = O(n \log(n)).$$

The expected running time of QuickSort is O(nlog(n)).

Proof:[*]

- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
    - The expected number of items on each side of the pivot is half of the things.
- If that occurs, the running time is $T(n) = O(n\log(n))$.
    - Since the relevant recurrence relation is $T(n) = 2T\left(\frac{n-1}{2}\right) + O(n)$
- Therefore, the expected running time is $O(n\log(n))$.

[*]Disclaimer: this proof is WRONG.

What’s wrong?

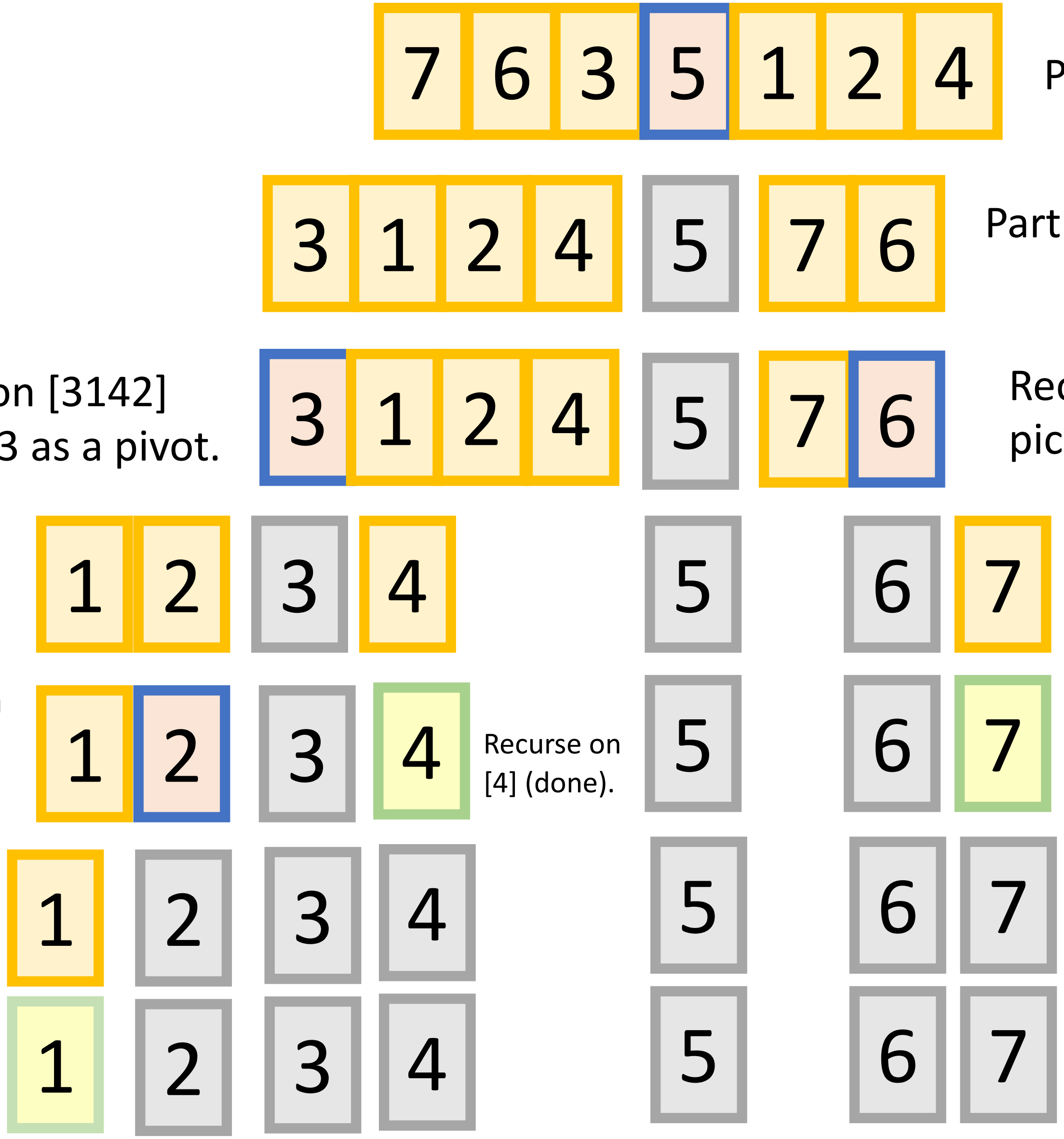
- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
  - The expected number of items on each side of the pivot is half of the things.
- If that occurs, the running time is $T(n) = O(n\log(n))$.
  - Since the relevant recurrence relation is $T(n) = 2T\left(\frac{n-1}{2}\right) + O(n)$
- Therefore, the expected running time is $O(n\log(n))$.

That’s not how expectations work!

Plucky the Pedantic Penguin

- The running time in the “expected” situation is not the same as the expected running time.
- Sort of like how $E[X^2]$ is not the same as $(E[X])^2$

Example of recursive calls



Pick 5 as a pivot

Partition on either side of 5

Recurse on [3142] and pick 3 as a pivot.

Recurse on [76] and pick 6 as a pivot.

Partition around 3.

Partition on either side of 6

Recurse on [12] and pick 2 as a pivot.

Recurse on [4] (done).

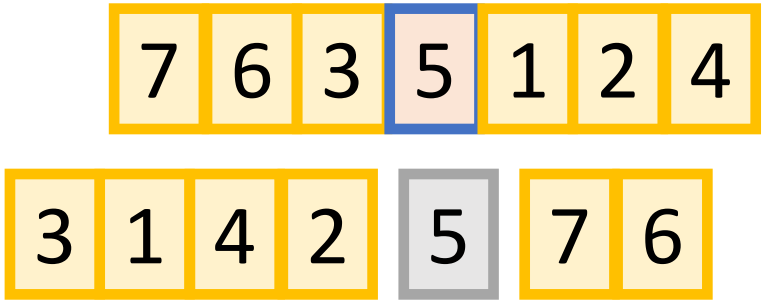
Recurse on [7], it has size 1 so we're done.

partition around 2.

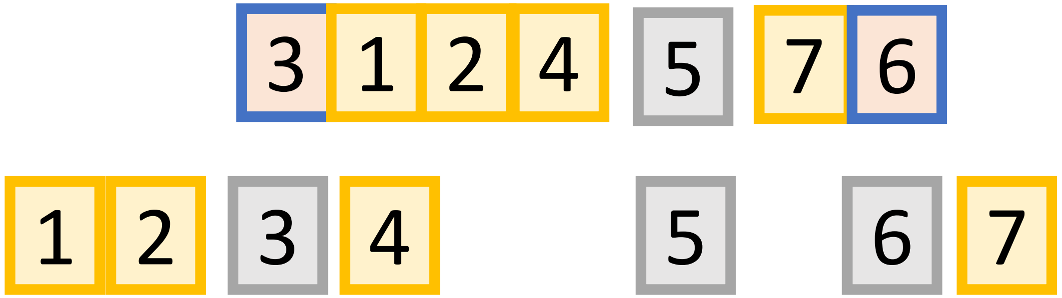
Recurse on [1] (done).

How long does this take to run?

- We will count the number of comparisons that the algorithm does.
  - This turns out to give us a good idea of the runtime. (Not obvious, but we can “charge” all operations to comparisons).
- How many times are any two items compared?

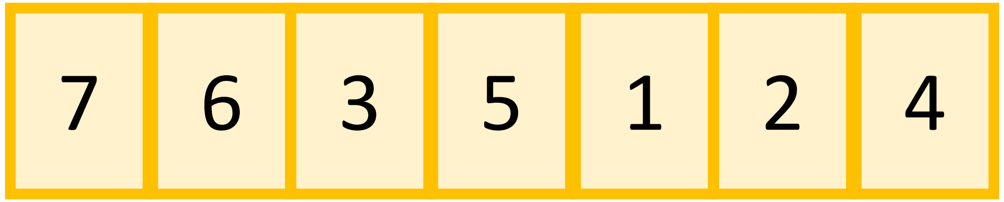


In the example before, everything was compared to 5 once in the first step….and never again.



But not everything was compared to 3.
5 was, and so were 1,2 and 4.
But not 6 or 7.

Each pair of items is compared either 0 or 1 times. Which is it?



Let's assume that the numbers in the array are actually the numbers 1,…,n

Of course this doesn't have to be the case! It's a good exercise to convince yourself that the analysis will still go through without this assumption.

- **Whether or not a, b are compared** is a random variable, that depends on the choice of pivots. Let's say

$$X_{a,b} = \begin{cases} 1 & \text{if } a \text{ and } b \text{ are ever compared} \\ 0 & \text{if } a \text{ and } b \text{ are never compared} \end{cases}$$

- In the previous example $X_{1,5} = 1$, because item 1 and item 5 were compared.
- But $X_{3,6} = 0$, because item 3 and item 6 were NOT compared.

Counting comparisons

- The number of comparisons total during the algorithm is

$$\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} X_{a,b}$$

- The expected number of comparisons is

$$E\left[\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} X_{a,b}\right] = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} E[\,X_{a,b}]$$

by using linearity of expectations.

Counting comparisons

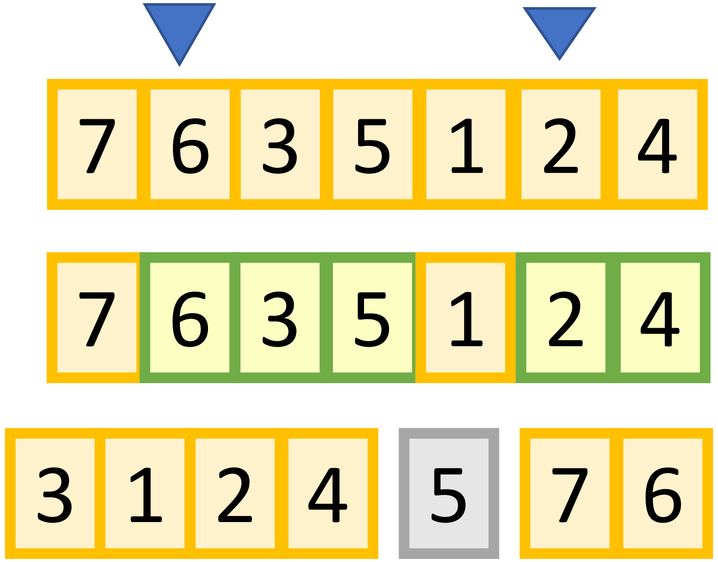
expected number of comparisons:

$$\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} E[\, X_{a,b} \,]$$

- So we just need to figure out $E[\ X_{a,b}\ ]$
- $E\left[X_{a,b}\right] = P(X_{a,b} = 1) \cdot 1 + P(X_{a,b} = 0) \cdot 0 = P(X_{a,b} = 1)$

  (by the definition of expectation)
- So we need to figure out:

$P(X_{a,b} = 1)$ = the probability that a and b are ever compared.



Say that a = 2 and b = 6. What is the probability that 2 and 6 are ever compared?

This is exactly the probability that either 2 or 6 is first picked to be a pivot out of the highlighted entries.

If, say, 5 were picked first, then 2 and 6 would be separated and never see each other again.

Counting comparisons

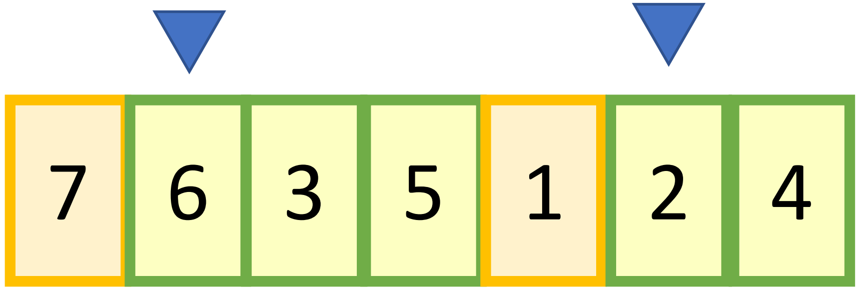
$P(\, X_{a,b} = 1 \,)$

= probability a,b are ever compared

= probability that one of a,b are picked first out of all of the b – a +1 numbers between them.

$$= \frac{2}{b - a + 1}$$

2 choices out of b-a+1…



All together now…

Expected number of comparisons

- $E\left[\sum_{a=1}^{n-1}\sum_{b=a+1}^{n} X_{a,b}\right]$

This is the expected number of comparisons throughout the algorithm

- $= \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} E[\, X_{a,b}]$

linearity of expectation

- $= \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} P(\, X_{a,b} = 1)$

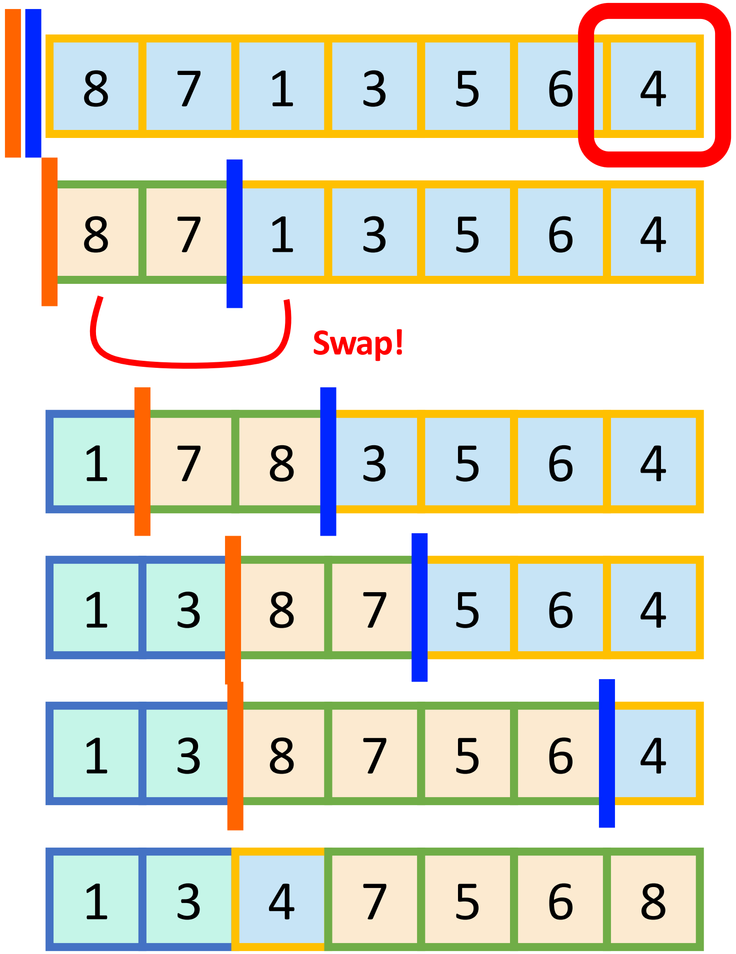
definition of expectation

- $= \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} \frac{2}{b-a+1}$

the reasoning we just did

- This is a big nasty sum, but we can do it.
- We get that this is less than 2n ln(n).

In-Place Partition for Quick Sort



Pivot

Choose it randomly, then swap it with the last one, so it's at the end.

Initialize 🟧 and 🟦

Step 🟦 forward.

When 🟦 sees something smaller than the pivot, **swap** the things ahead of the bars and increment both bars.

Repeat till the end, then put the pivot in the right place.

Practice example

- Input: n distinct ordered pairs of integers (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), where for all i, j, $x_i \neq x_j$ and $y_i \neq y_j$

- A set of points S is collinear if they all fall on the same line
  - That is, for all $(x_i, y_i) \in S,\ y_i = mx_i + b$ for fixed m and b

- Output: maximum integer N such that we can find N of the given points the same line

- Requirement: $O(n^2 \log n)$

Practice example

- For all pairs of points, compute their slope and intercept (m, b)
- QuickSort these pairs in increasing order of m, and then in increasing order of b as a tiebreaker.
- Iterate through the pairs, and note where the longest run of identical (m, b) pairs occurs
- Return a list of the points in this run of pairs

- Runtime:
    - there are $O(n^2)$ pairs of points
    - Quicksort takes $O(n^2 \log n^2) = O(n^2 \log n)$ time

Practice example

- CautiousQuickSort is the following:
  - the pivot is chosen by repeatedly randomly
  - stop if it partitions an array of n elements into two subarrays each with at least n/3 elements
    - Use partition to determine this condition
- Questions:
  - What is the probability of selecting a good pivot after a single trial?
    - 1/3
  - What is the maximum recursion depth of CautiousQuickSort?
    - O(log n)

Practice example

- What is the expected runtime?
  - Testing whether a pivot is good takes O(n) time
  - On each level of recursion, the expected number of random selections until a good pivot is found is 3
  - So, the expected amount of work done on each recursive level is O(n)
  - The depth is O(log n)
  - Thus, in total O(n log n) runtime

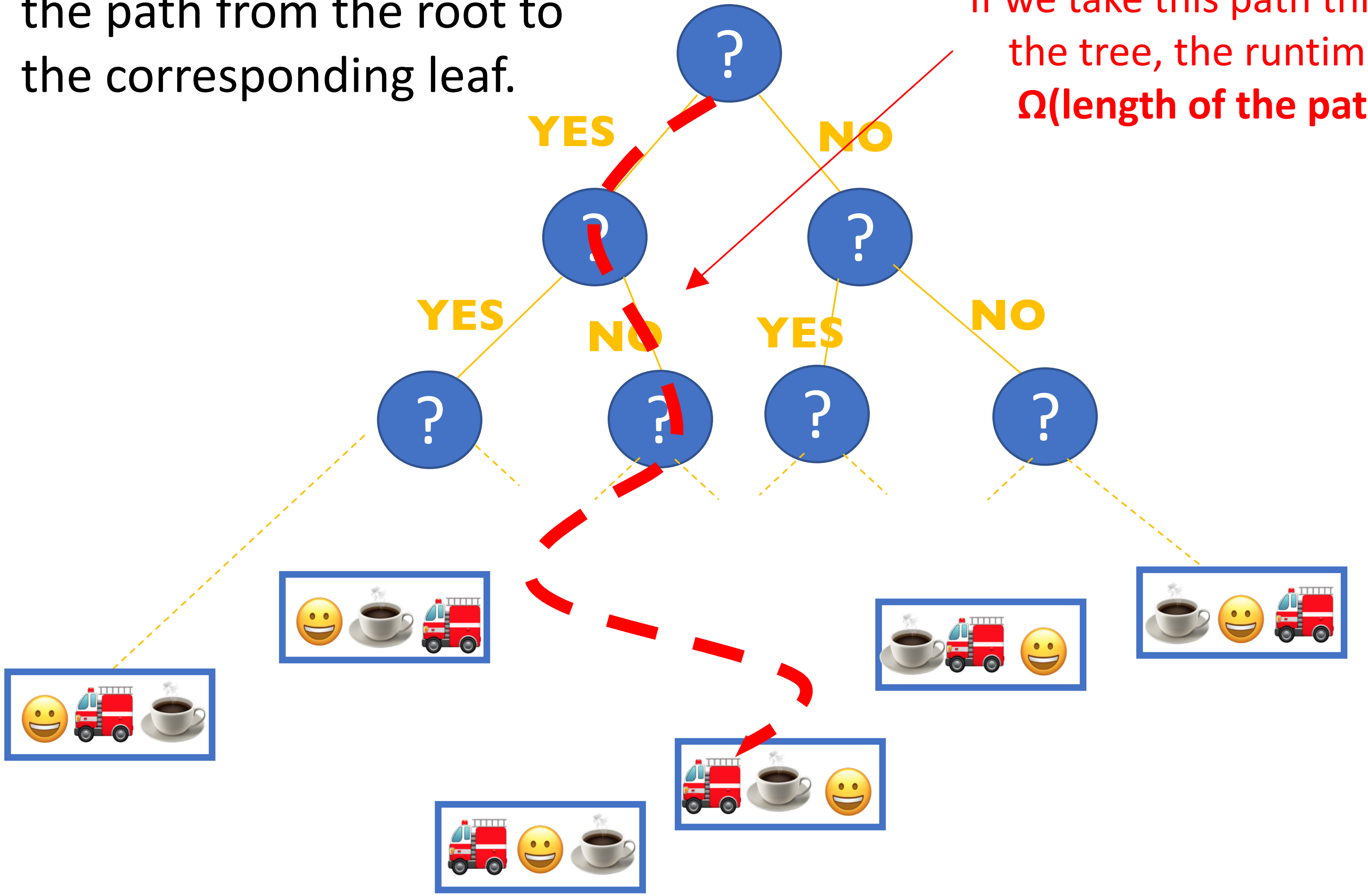
Lower bound for sorting

Lower bound of Ω(n log(n)).

- Theorem:
  - Any deterministic comparison-based sorting algorithm must take Ω(n log(n)) steps.

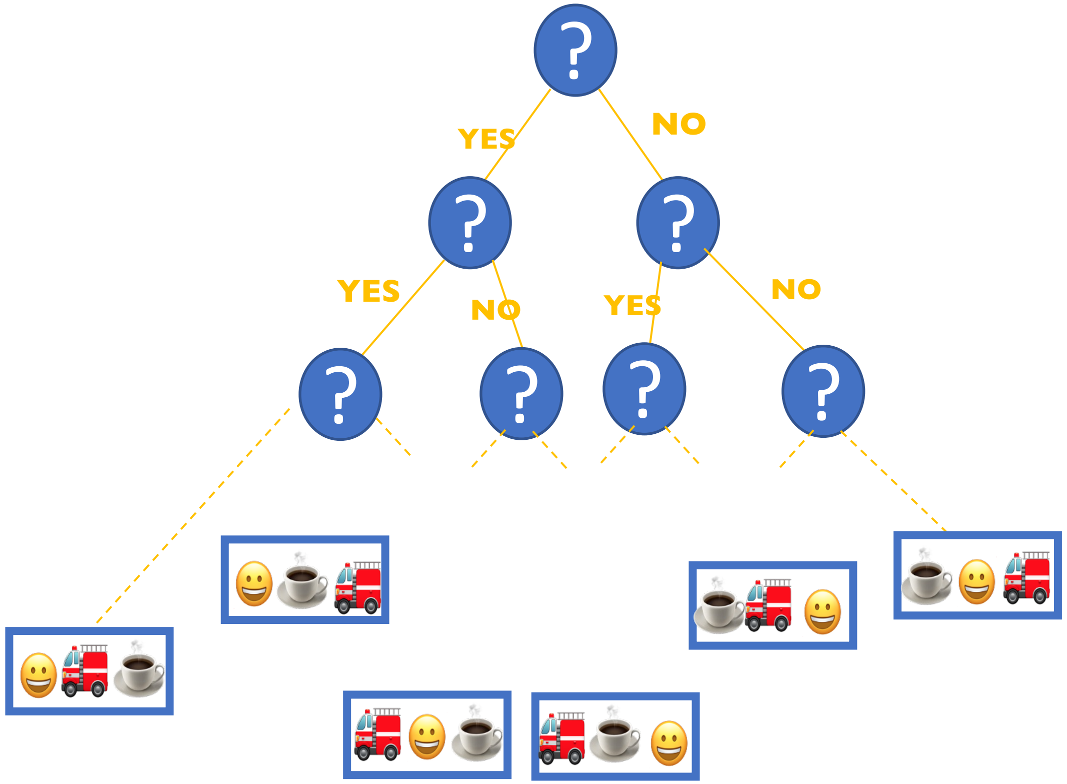
- Proof recap:
  - Any deterministic comparison-based algorithm can be represented as a decision tree with n! leaves.
  - The worst-case running time is at least the depth of the decision tree.
  - All decision trees with n! leaves have depth Ω(n log(n)).
  - So any comparison-based sorting algorithm must have worst-case running time at least Ω(n log(n)).

A: At least the length of the path from the root to the corresponding leaf.



If we take this path through the tree, the runtime is **Ω(length of the path).**

How long is the longest path?



We want a statement: in all such trees, the longest path is at least ______

- This is a binary tree with at least __n!__ leaves.
- The shallowest tree with n! leaves is the completely balanced one, which has depth _log(n!)_.
- So in all such trees, the longest path is at least log(n!).

- n! is about $(n/e)^n$ (Stirling's approximation).
- log(n!) is about n log(n/e) = Ω(n log(n)).

Conclusion: the longest path has length at least Ω(n log(n)).

Some “bad” news

- Theorem:
    - Any deterministic comparison-based sorting algorithm must take Ω(n log(n)) steps.
- Theorem:
    - Any randomized comparison-based sorting algorithm must take Ω(n log(n)) steps in expectation.

Bad Side: we can’t improve on n*log(n)

Bright Side: we know we’re done and can focus on other problems