طراحی الگوریتم ها (CE221)

جلسه هفتم:
انتخاب kامین عضو

سجاد شیرعلی شهرضا

بهار 1401

دوشنبه، 9 اسفند 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 4.3
- ارائه تمرین اول
    - مهلت ارسال تمرین اول : صبح شنبه، 14 اسفند 1400 (ساعت 8 صبح)

انتخاب kامین عضو

الگوریتم، اثبات درستی، زمان اجرا

THE SELECT PROBLEM

INPUT:

an unsorted array **A** of n elements (assume all elements are distinct),
& an integer **k** in {1, …, n}

7	2	6	9	1	5	4	11

OUTPUT of SELECT(A, k): the k^{th} smallest element of A

THE SELECT PROBLEM

INPUT:

an unsorted array **A** of n elements (assume all elements are distinct),
& an integer **k** in {1, …, n}

7	2	6	9	1	5	4	11

OUTPUT of SELECT(A, k): the k^{th} smallest element of A

SELECT(A, 1) = 1
SELECT(A, 2) = 2
SELECT(A, 3) = 4
SELECT(A, 8) = 11

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

Note: k is a 1-indexed number!

THE SELECT PROBLEM

INPUT:

an unsorted array **A** of n elements (assume all elements are distinct),
& an integer **k** in {1, …, n}

7	2	6	9	1	5	4	11

OUTPUT of SELECT(A, k): the k^{th} smallest element of A

Can you come up with an O(n log n) algorithm for SELECT?

AN O(n log n) ALGORITHM

```
SELECT(A,k):
    A = MERGESORT(A)
    return A[k-1]
```

It's k-1 (rather than k) since my pseudocode is 0-indexed and k is a 1-indexed number

Okay, great! We're done!

سوال؟

AN O(n log n) ALGORITHM

SE

THE QUESTION IS…

CAN WE DO BETTER?

It’s k-1 (rather than k) since my pseudocode is 0-indexed and k is a 1-indexed number

~~Okay, great! We’re done!~~

GOAL: AN O(n) ALGORITHM

If k = 1, then we want the minimum of A. There's an easy O(n) algorithm for that:

Pretty much the same if k = n (we're just finding MAX(A) instead)

GOAL: AN O(n) ALGORITHM

If k = 1, then we want the minimum of A. There’s an easy O(n) algorithm for that:

Pretty much the same if k = n (we’re just finding MAX(A) instead)

```
SELECT-1(A):
    result = infinity
    for i in [0,...,n-1]:
        if A[i] < result:
            result = A[i]
    return result
```

This loop runs O(n) times

The body of each iteration is O(1) work.

Runtime of SELECT-1: O(n)

GOAL: AN O(n) ALGORITHM

If k = 2, then we want the second-smallest element in A.

There's an easy-ish O(n) algorithm for that:

(Not a very important algorithm, because this will end up being a bad idea…)

GOAL: AN O(n) ALGORITHM

If k = 2, then we want the second-smallest element in A.
There's an easy-ish O(n) algorithm for that:

(Not a very important algorithm, because this will end up being a bad idea…)

```
SELECT-2(A):
    result = infinity
    minSoFar = infinity
    for i in [0,...,n-1]:
        if A[i] < result & A[i] < minSoFar:
            result = minSoFar
            minSoFar = A[i]
        else if A[i] < result & A[i] >= minSoFar
            result = A[i]
    return result
```

This loop runs O(n) times

The body of each iteration is still O(1) work.

Runtime of SELECT-2: O(n)

GOAL: AN O(n) ALGORITHM

If k = n/2, then we want the median element in A.

```
SELECT-n/2(A):
    result = infinity
    minSoFar = infinity
    secondMinSoFar = infinity
    thirdMinSoFar = infinity
    fourthMinSoFar = infinity
    fifthMinSoFar = infinity
    ...
```

GOAL: AN O(n) ALGORITHM

If k = n/2, then we want the median element in A.

```
SELECT-n/2(A):
    result = infinity
    minSoFar = infinity
    secondMinSoFar = infinity
    thirdMinSoFar = infinity
    fourthMinSoFar = infinity
    fifthMinSoFar = infinity
    ...
```

Runtime of SELECT-n/2: O(n^2)

Clearly, this algorithm style isn't a good idea for large k (e.g. n/2).
This basically ends up looking like InsertionSort.

LINEAR SELECTION: THE IDEA

Let’s use DIVIDE-and-CONQUER!

LINEAR SELECTION: THE IDEA

Let’s use DIVIDE-and-CONQUER!

Select a pivot

Partition around it

Recurse!

LINEAR SELECTION: THE IDEA

Let’s use DIVIDE-and-CONQUER!

Select a pivot

Partition around it

Recurse!

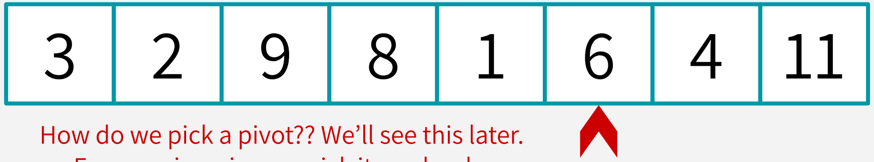
kind of like a “binary search” for the kth smallest element (except that the array isn’t sorted!)

LINEAR SELECTION: THE IDEA

3	2	9	8	1	6	4	11

LINEAR SELECTION: THE IDEA

Select a pivot

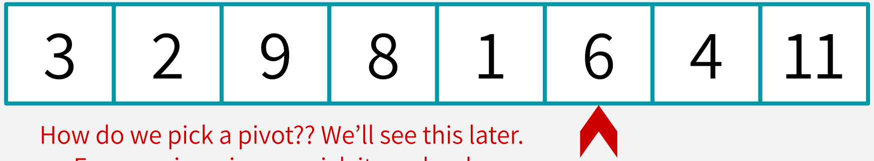


3	2	9	8	1	6	4	11

How do we pick a pivot?? We’ll see this later.
For now, imagine we pick it randomly.

LINEAR SELECTION: THE IDEA

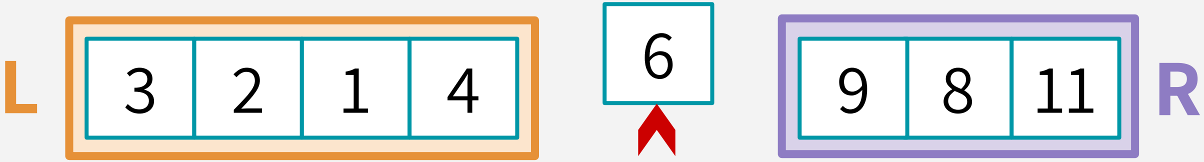
Select a pivot



3	2	9	8	1	6	4	11

How do we pick a pivot?? We’ll see this later.
For now, imagine we pick it randomly.

Partition around it



L

3	2	1	4

6

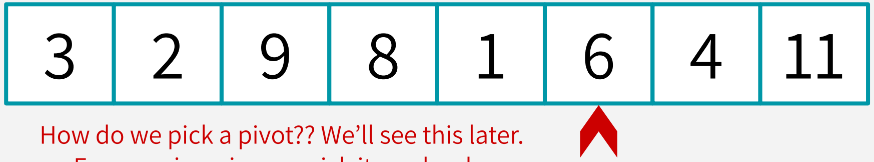
9	8	11

R

Partition around pivot: **L** has elements less than pivot, and **R** has elements greater than pivot.
(Note that **L** and **R** remain unsorted).

LINEAR SELECTION: THE IDEA

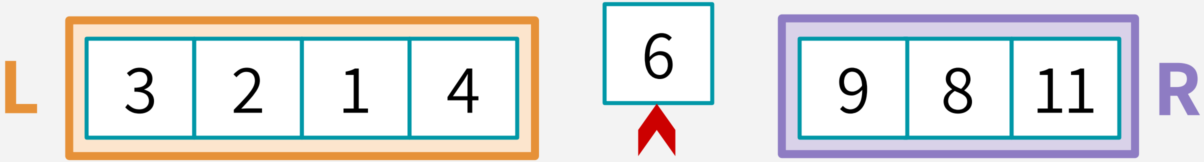
Select a pivot



3	2	9	8	1	6	4	11

How do we pick a pivot?? We'll see this later.
For now, imagine we pick it randomly.

Partition around it



3	2	1	4

6

9	8	11

Partition around pivot: **L** has elements less than pivot, and **R** has elements greater than pivot.
(Note that **L** and **R** remain unsorted).

Recurse!

The pivot is in position **5**. We have three cases:

1. **if k = 5: return pivot** — the kth smallest element is the pivot!
2. **if k < 5: return SELECT(L, k)** — the kth smallest element lives in L
3. **if k > 5: return SELECT(R, k-5)** — the kth smallest element is the (k-5)th smallest element in R

LINEAR SELECTION: EXAMPLE

SELECT(A, 7):

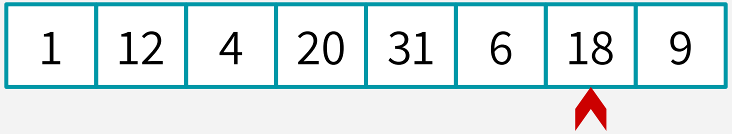
1	12	4	20	31	6	18	9

LINEAR SELECTION: EXAMPLE

SELECT`(A, 7):`

PICK A PIVOT

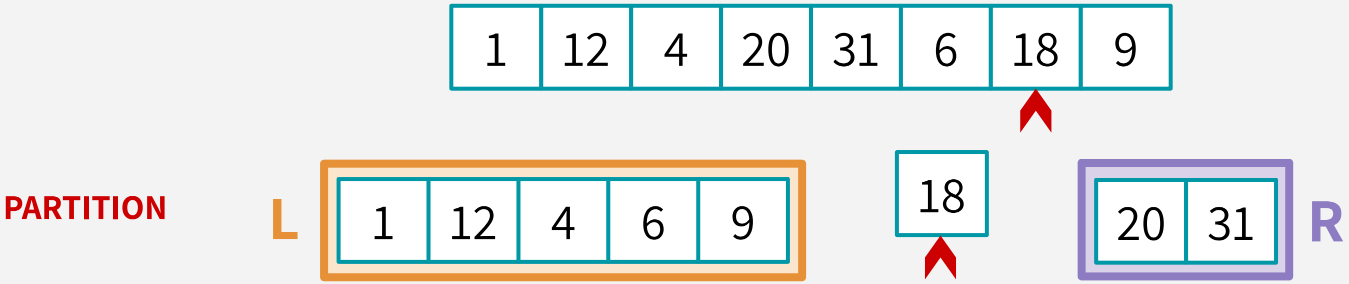
How do we pick a pivot???
We'll see later...



1	12	4	20	31	6	18	9

LINEAR SELECTION: EXAMPLE

SELECT(A, 7):



1	12	4	20	31	6	18	9

PARTITION

L

1	12	4	6	9

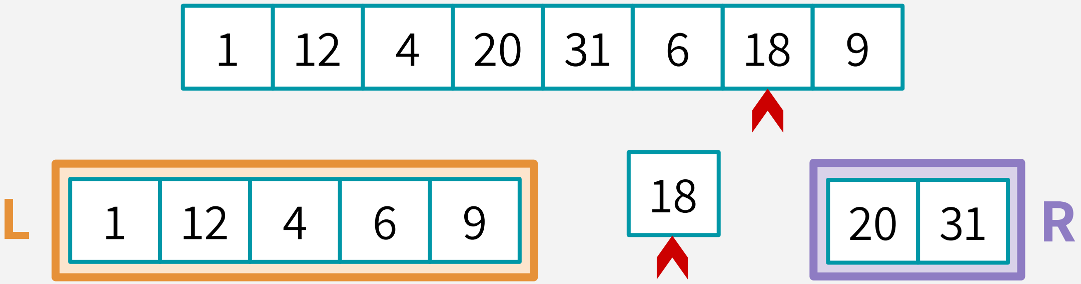
18

20	31

R

LINEAR SELECTION: EXAMPLE

SELECT(A, 7):



1	12	4	20	31	6	18	9

L

1	12	4	6	9

18

20	31

R

Recurse here (since 18 occupies index 6 and k = 7 > 6)

RECURSE

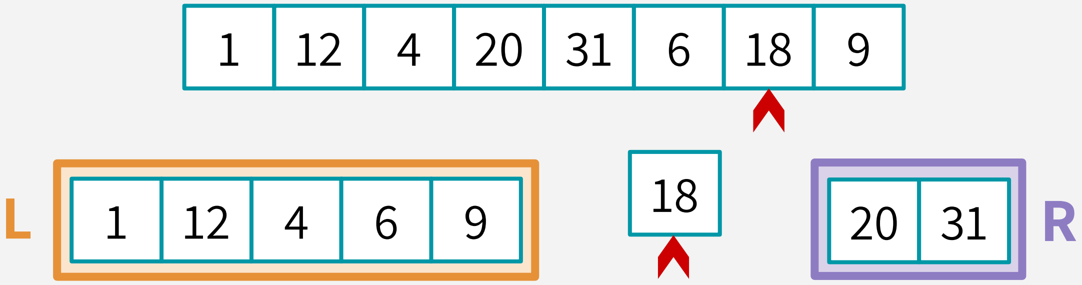
SELECT(R, 1):

20	31

1 = 7 - 6
(aka k minus pivot position)

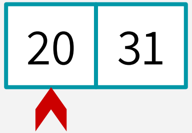
LINEAR SELECTION: EXAMPLE

SELECT`(A, 7):`



Recurse here (since 18 occupies index 6 and k = 7 > 6)

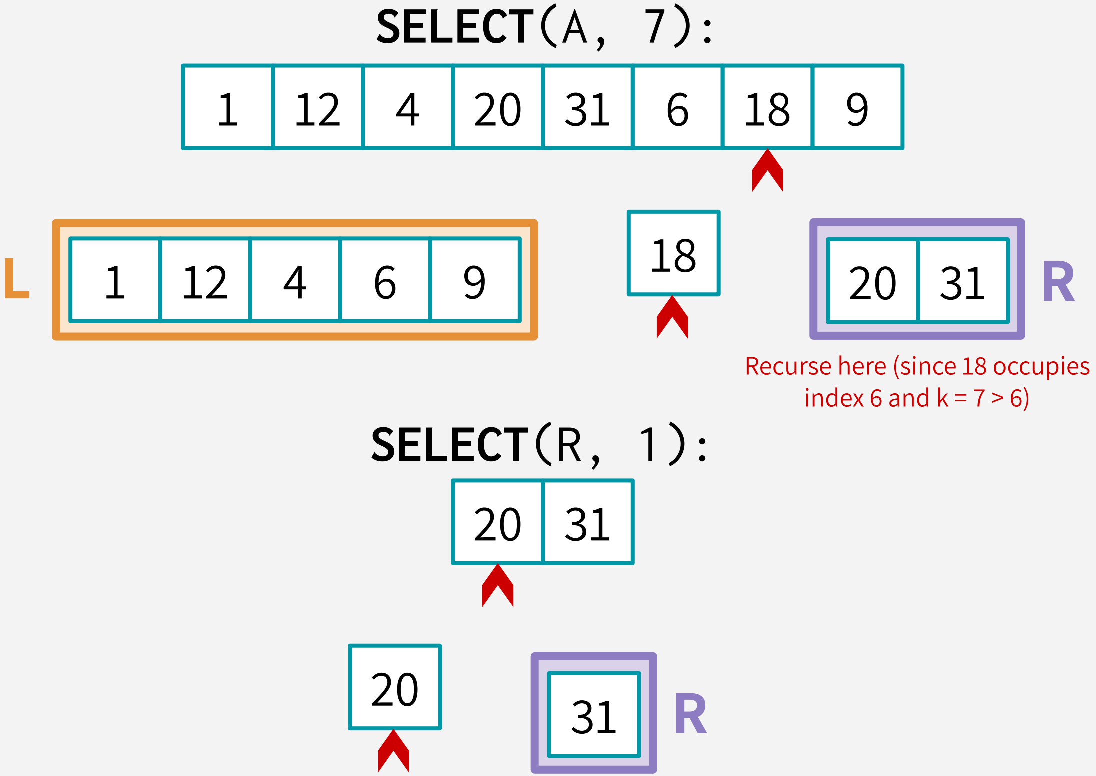
SELECT`(R, 1):`



PICK A PIVOT

How do we pick a pivot???
We'll see later...

LINEAR SELECTION: EXAMPLE



SELECT(A, 7):

1	12	4	20	31	6	18	9

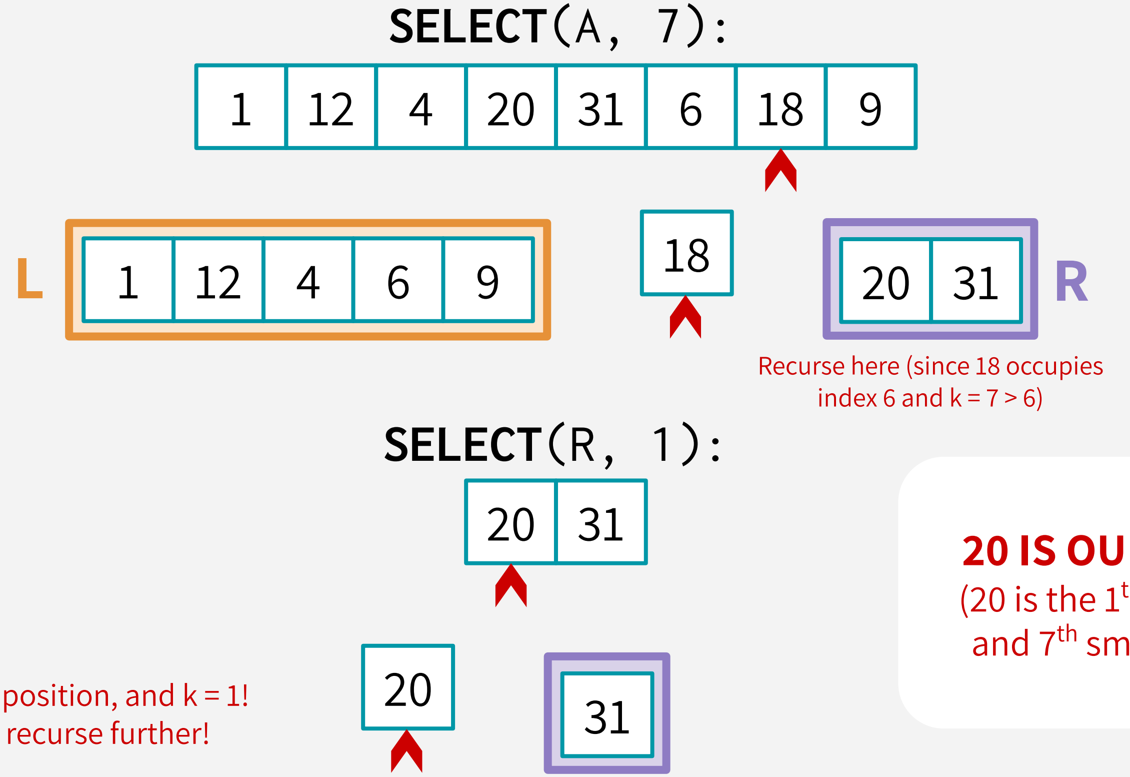
Recurse here (since 18 occupies index 6 and k = 7 > 6)

SELECT(R, 1):

20	31

PARTITION

LINEAR SELECTION: EXAMPLE



SELECT(A, 7):

1	12	4	20	31	6	18	9

L

1	12	4	6	9

18

20	31

R

Recurse here (since 18 occupies index 6 and k = 7 > 6)

SELECT(R, 1):

20	31

20

31

20 is in the 1th position, and k = 1!
No need to recurse further!

20 IS OUR ANSWER!
(20 is the 1th smallest in R, and 7th smallest overall)

LINEAR SELECTION: PSEUDOCODE

Base Case:
if len(A) = 1, then just go ahead and return the element itself

```
SELECT(A,k):
    if len(A) == 1:
        return A[0]
    p = GET_PIVOT(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
        return p
    else if len(L) > k-1:
        return SELECT(L, k)
    else:
        return SELECT(R, k-len(L)-1)
```

Case 1:
We got lucky and found exactly the k^{th} smallest!

Case 2:
The k^{th} smallest is in the first part of the array (L)

Case 3:
The k^{th} smallest is in the second part of the array (R)

LINEAR SELECTION: PSEUDOCODE

```
SELECT(A,k):
    if len(A) == 1:
        return A[0]
    p = GET_PIVOT(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
        return p
    else if len(L) > k-1:
        return SELECT(L, k)
    else:
        return SELECT(R, k-len(L)-1)
```

```
PARTITION(A, pivot):
    L, R = [], []
    for i in [1,...,len(A)]:
        if A[i] == pivot:
            continue
        else if A[i] < pivot:
            add A[i] to L
        else:
            add A[i] to R
```

سوال؟

LINEAR SELECTION: SO FAR

- Intuition:
  - Partition the array around a pivot (how do we select?? still TBD)
  - Either return the pivot itself or recurse on the left or right subarrays (but not both!)

```
SELECT(A,k):
    if len(A) == 1:
         return A[0]
    p = GET_PIVOT(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:      return p
    else if len(L) > k-1:  return SELECT(L, k)
    else:                  return SELECT(R, k-len(L)-1)
```

LINEAR SELECTION: SO FAR

- Intuition:
  - Partition the array around a pivot (how do we select?? still TBD)
  - Either return the pivot itself or recurse on the left or right subarrays (but not both!)

- Our two favorite questions:
  - Does this work?
  - What’s the runtime?

```
SELECT(A,k):
    if len(A) == 1:
          return A[0]
    p = GET_PIVOT(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:     return p
    else if len(L) > k-1: return SELECT(L, k)
    else:                 return SELECT(R, k-len(L)-1)
```

LINEAR SELECTION: DOES IT WORK?

FROM PREVIOUS WEEKS!

RECURSIVE ALGORITHMS

1. **Inductive hypothesis**: your algorithm is correct for sizes *up to* **i**
2. **Base case**: IH holds for i < small constant
3. **Inductive step**:
   - assume IH holds for k ⇒ prove k+1, *OR*
   - assume IH holds for {1,2,...,k-1} ⇒ prove k.
4. **Conclusion**: IH holds for i = n ⇒ yay!

INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

When run on an array A of size **i** and an integer 1 ≤ k ≤ **i**, SELECT(A,k) correctly returns the kth smallest element of A.

INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

When run on an array A of size **i** and an integer 1 ≤ k ≤ **i**, SELECT(A,k) correctly returns the kth smallest element of A.

BASE CASE

The IH holds for i = 1: We know k must be 1, so SELECT does indeed return the smallest (and only) element of A.

INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

When run on an array A of size **i** and an integer $1 \le k \le$ **i**, SELECT(A,k) correctly returns the k^{th} smallest element of A.

BASE CASE

The IH holds for $i = 1$: We know k must be 1, so SELECT does indeed return the smallest (and only) element of A.

(OUTLINE OF) INDUCTIVE STEP *(strong/complete induction)*

Let j be an integer, where $j > 1$. Assume that the IH holds for all i where $1 \le i < j$. We want to show that the IH holds for $i = j$, i.e. that for an array A of size j and an integer $k \le j$, SELECT returns the k^{th} smallest element of A.

We consider three cases, depending on the pivot chosen by GET_PIVOT. PARTITION gives us L, and R.

- **CASE 1**: $|L| = k-1$.
- **CASE 2**: $|L| > k-1$.
- **CASE 3**: $|L| < k-1$.

We use STRONG induction because cases 2 and 3 rely on the correctness of the smaller recursive calls.

Thus, in each of the three cases, SELECT(A,k) returns the k^{th} smallest element of A. This establishes the IH for $i = j$.

INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

When run on an array A of size **i** and an integer $1 \le k \le$ **i**, SELECT(A,k) correctly returns the k^{th} smallest element of A.

BASE CASE

The IH holds for $i = 1$: We know k must be 1, so SELECT does indeed return the smallest (and only) element of A.

(OUTLINE OF) INDUCTIVE STEP *(strong/complete induction)*

Let j be an integer, where $j > 1$. Assume that the IH holds for all i where $1 \le i < j$. We want to show that the IH holds for $i = j$, i.e. that for an array A of size j and an integer $k \le j$, SELECT returns the k^{th} smallest element of A.

We consider three cases, depending on the pivot chosen by GET_PIVOT. PARTITION gives us L, and R.

- **CASE 1**: $|L| = k-1$.
- **CASE 2**: $|L| > k-1$.
- **CASE 3**: $|L| < k-1$.

We use STRONG induction because cases 2 and 3 rely on the correctness of the smaller recursive calls.

Thus, in each of the three cases, SELECT(A,k) returns the k^{th} smallest element of A. This establishes the IH for $i = j$.

CONCLUSION

By induction, we conclude that the IH holds for all $1 \le i \le n$. Thus, we conclude that SELECT(A, k) returns the k^{th} smallest element of A on any array A, provided that $1 \le k \le |A|$. That is, SELECT is correct!

سوال؟

RUNTIME

```
SELECT(A,k):
   if len(A) == 1:
       return A[0]
   p = GET_PIVOT(A)
   L, R = PARTITION(A,p)
   if len(L) == k-1:
       return p
   else if len(L) > k-1:
       return SELECT(L, k)
   else if len(L) < k-1:
       return SELECT(R, k-len(L)-1)
```

Recurrence Relation for SELECT

For now, assume we’ll pick the pivot in time O(n)

RUNTIME

```
SELECT(A,k):
    if len(A) == 1:
         return A[0]
    p = GET_PIVOT(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
         return p
    else if len(L) > k-1:
         return SELECT(L, k)
    else if len(L) < k-1:
         return SELECT(R, k-len(L)-1)
```

Recurrence Relation for SELECT

For now, assume we'll pick the pivot in time O(n)

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

RUNTIME

```
SELECT(A,k):
   if len(A) == 1:
        return A[0]
   p = GET_PIVOT(A)
   L, R = PARTITION(A,p)
   if len(L) == k-1:
        return p
   else if len(L) > k-1:
        return SELECT(L, k)
   else if len(L) < k-1:
        return SELECT(R, k-len(L)-1)
```

Recurrence Relation for SELECT

For now, assume we'll pick the pivot in time O(n)

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

But what are `len(L)` and `len(R)`?
That depends on how we pick the pivot...

RUNTIME

```
SELECT(A,k):
    if len(A) == 1:
         return A[0]
    p = GET_PIVOT(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
         return p
    else if len(L) > k-1:
         return SELECT(L, k)
    else if len(L) < k-1:
         return SELECT(R, k-len(L)-1)
```

What’s a “good” pivot?
What’s a “bad” pivot?

e Relation for SELECT

we’ll pick the pivot in time O(n)

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

But what are `len(L)` and `len(R)`?
That depends on how we pick the pivot...

THE WORST PIVOT

The WORST pivot: picking the max or the min each time!

Then, in the worst case, the recurrence relation looks like T(n) = T(n-1) + O(n).

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

THE WORST PIVOT

The WORST pivot: picking the max or the min each time!

Then, in the worst case, the recurrence relation looks like T(n) = T(n-1) + O(n).

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

This ends up being $\Omega(n^2)$!

A call to SELECT(A, n/2) would already consist of ~n/2 recursive calls (each with a subarray of length at least n/2)!

THE IDEAL PIVOT

The IDEAL pivot: splits the input array exactly in half!

`len(L)` = `len(R)` = (n-1)/2

$$T(n) = \begin{cases} O(n) & \texttt{len(L) == k-1} \\ T(\texttt{len(L)}) + O(n) & \texttt{len(L) > k-1} \\ T(\texttt{len(R)}) + O(n) & \texttt{len(L) < k-1} \end{cases} \quad \Rightarrow \quad \mathbf{T(n) \le T(n/2) + O(n)}$$

THE IDEAL PIVOT

The IDEAL pivot: splits the input array exactly in half!

$\texttt{len(L)} = \texttt{len(R)} = (n\text{-}1)/2$

$$T(n) = \begin{cases} O(n) & \texttt{len(L) == k-1} \\ T(\texttt{len(L)}) + O(n) & \texttt{len(L) > k-1} \\ T(\texttt{len(R)}) + O(n) & \texttt{len(L) < k-1} \end{cases} \Longrightarrow \mathbf{T(n) \leq T(n/2) + O(n)}$$

a = 1
b = 2
d = 1

$\mathbf{a < b^d}$

Suppose $\mathbf{T(n) = a \cdot T(n/b) + O(n^d)}$. The Master Theorem states:

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

THE IDEAL PIVOT

The IDEAL pivot: splits the input array exactly in half!

`len(L` n-1)/2

$$T(n) = \begin{cases} O(n) & \texttt{le} \\ T(\texttt{len(L)}) + O(n) & \texttt{l} \\ T(\texttt{len(R)}) + O(n) & \texttt{l} \end{cases}$$

With the ideal pivot, the runtime would be:

O(n)

$$\mathbf{T(n) \leq T(n/2) + O(n)}$$

a = 1
b = 2
d = 1

$$\mathbf{a < b^d}$$

Suppose **T(n) = a · T(n/b)** Master Theorem states:

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

THE IDEAL PIVOT

The IDEAL pivot: splits the input array exactly in half!

Sadly, the pivot to divide the input in half is the

MEDIAN

aka ***SELECT(A, n/2)***

aka exactly the problem we're trying to solve…

$$T(n) = \begin{cases} O(r \\ T(1 \\ T(1 \end{cases}$$

+ O(n)

$\mathbf{b^d}$

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

سوال؟

THE GOOD-ENOUGH PIVOT

The GOOD-ENOUGH pivot: splits the input array kind of in half!

3n/10 < `len(L)` < 7n/10

3n/10 < `len(R)` < 7n/10

THE GOOD-ENOUGH PIVOT

The GOOD-ENOUGH pivot: splits the input array kind of in half!

$3n/10 < \texttt{len(L)} < 7n/10$

$3n/10 < \texttt{len(R)} < 7n/10$

If we could fetch this good-enough pivot in time O(n), let's say, the recurrence looks like:

$$T(n) = \begin{cases} O(n) & \texttt{len(L) == k-1} \\ T(\texttt{len(L)}) + O(n) & \texttt{len(L) > k-1} \\ T(\texttt{len(R)}) + O(n) & \texttt{len(L) < k-1} \end{cases} \quad \Longrightarrow \quad \mathbf{T(n) \leq T(7n/10) + O(n)}$$

THE GOOD-ENOUGH PIVOT

The GOOD-ENOUGH pivot: splits the input array kind of in half!

$$3n/10 < \texttt{len(L)} < 7n/10$$

$$3n/10 < \texttt{len(R)} < 7n/10$$

If we could fetch this good-enough pivot in time O(n), let’s say, the recurrence looks like:

$$T(n) = \begin{cases} O(n) & \texttt{len(L) == k-1} \\ T(\texttt{len(L)}) + O(n) & \texttt{len(L) > k-1} \\ T(\texttt{len(R)}) + O(n) & \texttt{len(L) < k-1} \end{cases} \quad \Rightarrow \quad \mathbf{T(n) \leq T(7n/10) + O(n)}$$

$$\mathbf{a = 1}$$
$$\mathbf{b = 10/7} \qquad \mathbf{a < b^d}$$
$$\mathbf{d = 1}$$

Suppose $\mathbf{T(n) = a \cdot T(n/b) + O(n^d)}$. The Master Theorem states:

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

THE GOOD-ENOUGH PIVOT

The GOOD-ENOUGH pivot: splits the input array kind of in half!

3n/10 < len(L) < 7n/10

3n/ /10

If we could fetch this good-enoug t's say, the recurrence looks like:

$$T(n) = \begin{cases} O(n) & \texttt{l} \\ T(\texttt{len(L)}) + O(n) & \texttt{l} \\ T(\texttt{len(R)}) + O(n) & \texttt{le} \end{cases}$$

This good-enough pivot would still give us:

O(n)

$$\mathbf{T(n) \leq T(7n/10) + O(n)}$$

a = 1
b = 10/7
d = 1

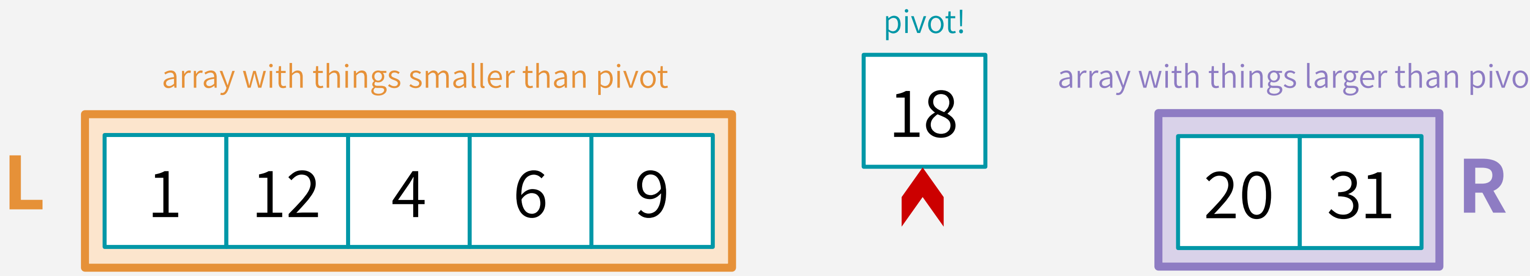
$\mathbf{a < b^d}$

Suppose $\mathbf{T(n) = a \cdot T(n/b)}$ Master Theorem states:

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

OUR GOAL

Efficiently pick the pivot in time O(n) so that



pivot!

array with things smaller than pivot

array with things larger than pivot

L

1	12	4	6	9

18

20	31

R

$\mathbf{3n/10} < \texttt{len(L)} < \mathbf{7n/10}$ $\mathbf{3n/10} < \texttt{len(R)} < \mathbf{7n/10}$

Then, our recurrence $T(n) \leq T(7n/10) + O(n)$ comes out to **O(n)**!

سوال؟

میانه ی میانه ها!

ایده اصلی الگوریتم خطی برای انتخاب kامین عضو

MEDIAN-OF-MEDIANS

The ideal world wasn't feasible because we can't just compute SELECT(A, n/2) ⇒ that would throw us into infinite recursion since problem sizes aren't shrinking between recursive calls…

But we can instead generate a ***smaller*** list and call SELECT on that smaller list!

MEDIAN-OF-MEDIANS

The ideal world wasn't feasible because we can't just compute SELECT(A, n/2) ⇒ that would throw us into infinite recursion since problem sizes aren't shrinking between recursive calls…

But we can instead generate a ***smaller*** list and call SELECT on that smaller list!

OUR GAME PLAN:

We'll make a smaller list out of SUB-MEDIANS.

Then, we'll use SELECT to find the median of the sub-medians.

This "median of medians" will be our proxy for the true median!

MEDIAN-OF-MEDIANS

GOAL: get a proxy for the true median by finding the exact median of all the sub-medians!

1	14	4	18	25	6	17	9	3	5	10	16	12	23	19	13	20	8	15	24	7	21	22	2	11

MEDIAN-OF-MEDIANS

GOAL: get a proxy for the true median by finding the exact median of all the sub-medians!

Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)

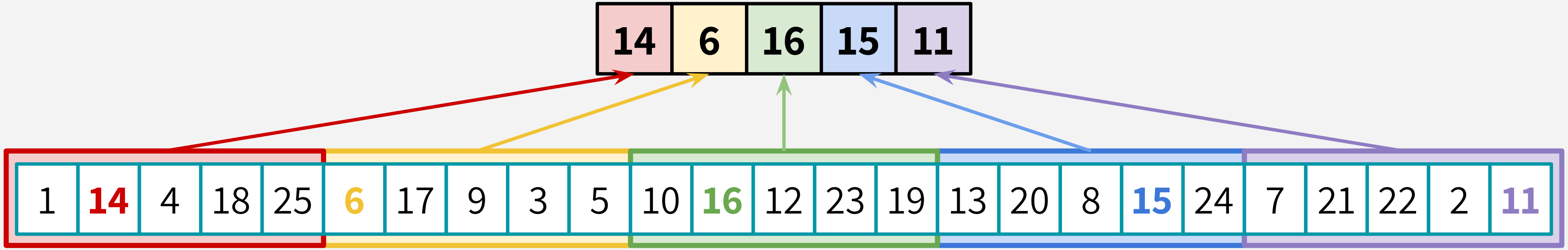
1	14	4	18	25	6	17	9	3	5	10	16	12	23	19	13	20	8	15	24	7	21	22	2	11

MEDIAN-OF-MEDIANS

GOAL: get a proxy for the true median by finding the exact median of all the sub-medians!

Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)

Find the sub-median of each small group (3rd smallest out of the 5)



14	**6**	**16**	**15**	**11**

1	**14**	4	18	25	**6**	17	9	3	5	10	**16**	12	23	19	13	20	8	**15**	24	7	21	22	2	**11**

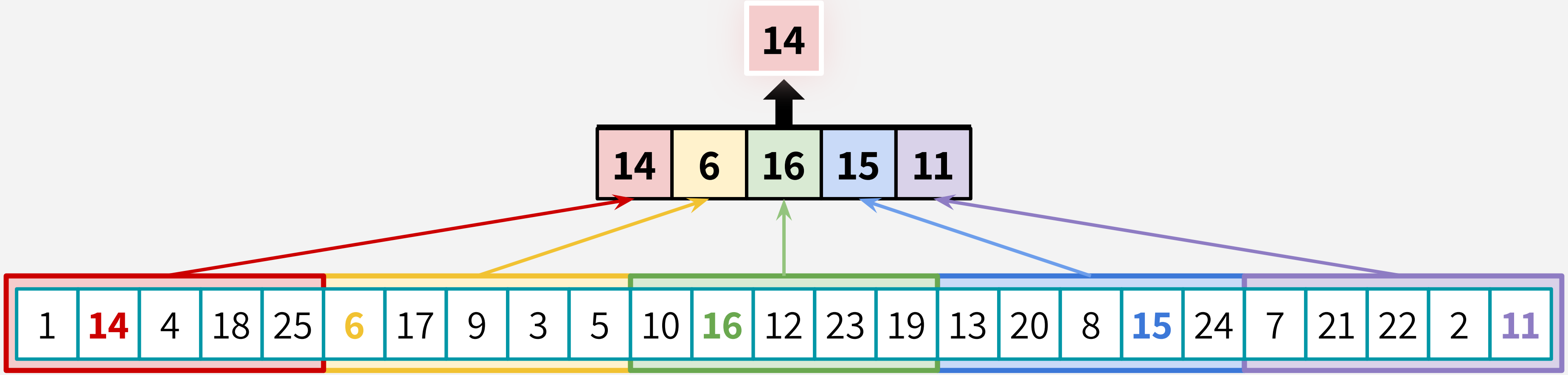
MEDIAN-OF-MEDIANS

GOAL: get a proxy for the true median by finding the exact median of all the sub-medians!

Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)

Find the sub-median of each small group (3rd smallest out of the 5)

Find the median of all the sub-medians (call SELECT)



MEDIAN-OF-MEDIANS

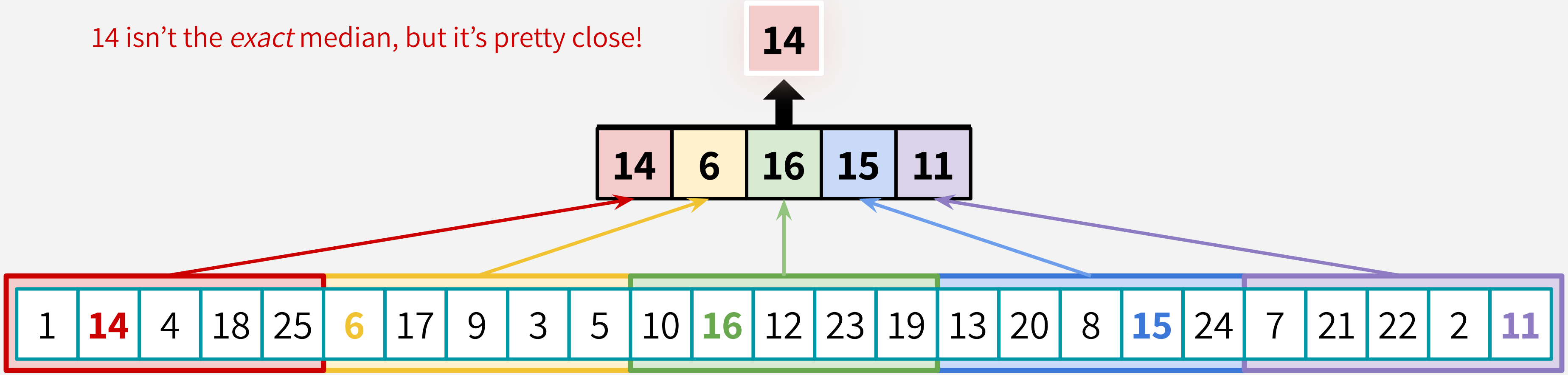
GOAL: get a proxy for the true median by finding the exact median of all the sub-medians!

Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)

Find the sub-median of each small group (3rd smallest out of the 5)

Find the median of all the sub-medians (call SELECT)

14 isn't the *exact* median, but it's pretty close!



MEDIAN-OF-MEDIANS

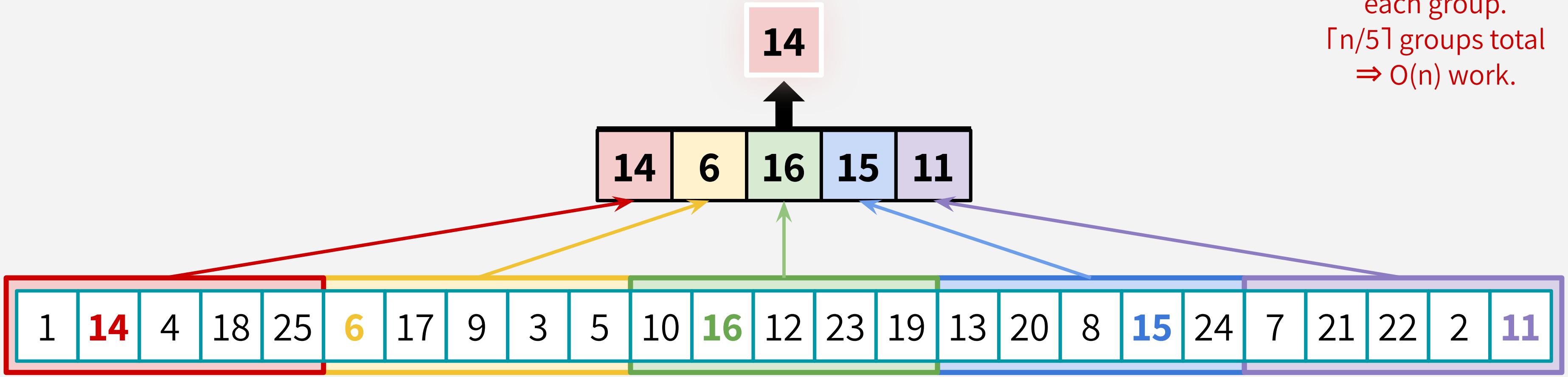
GOAL: get a proxy for the true median by finding the exact median of all the sub-medians!

Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)

Find the sub-median of each small group (3rd smallest out of the 5)

Find the median of all the sub-medians (call SELECT)

constant work for each group. ⌈n/5⌉ groups total ⇒ O(n) work.



14

14	6	16	15	11

1	14	4	18	25	6	17	9	3	5	10	16	12	23	19	13	20	8	15	24	7	21	22	2	11

MEDIAN-OF-MEDIANS

GOAL: get a proxy for the true median by finding the exact median of all the sub-medians!

Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)

Find the sub-median of each small group (3rd smallest out of the 5)

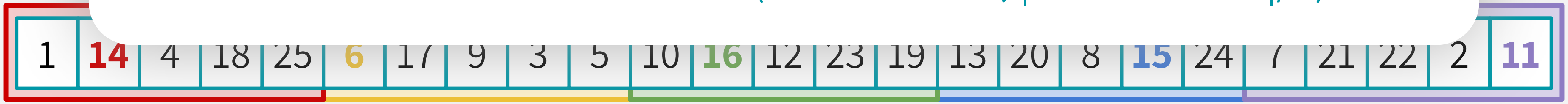
Find the median of all the sub-medians (call SELECT)

constant work for each group. ⌈n/5⌉ groups total ork.

14

To compute our pivot:

Do O(n) work to set up (divide into groups & get a list of submedians), then make a call to **SELECT**(Submedians, |Submedians|/2)



1	**14**	4	18	25	**6**	17	9	3	5	10	**16**	12	23	19	13	20	8	**15**	24	7	21	22	2	**11**

سوال؟

ANALYZING RUNTIME

```
SELECT(A,k):
    if len(A) == 1:
        return A[0]
    p = MEDIAN_OF_MEDIANS(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
        return p
    else if len(L) > k-1:
        return SELECT(L, k)
    else:
        return SELECT(R, k-len(L)-1)
```

What does the recurrence relation for T(n) look like?

ANALYZING RUNTIME

```
SELECT(A,k):
    if len(A) == 1:
        return A[0]
    p = MEDIAN_OF_MEDIANS(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
        return p
    else if len(L) > k-1:
        return SELECT(L, k)
    else:
        return SELECT(R, k-len(L)-1)
```

O(n) work outside of recursive calls

(base case, set-up within MEDIAN_OF_MEDIANS, partitioning)

T(n/5) work hidden in this recursive call

(remember, MEDIAN_OF_MEDIANS calls SELECT on ⌈n/5⌉-size array)

T(???) work hidden in this recursive call

What is the maximum size of either L or R?

ANALYZING RUNTIME

```
SELECT(A,k):
    if len(A) == 1:
    else:
        return SELECT(R, k-len(L)-1)
```

What is the smallest number of elements that could be smaller than our MEDIAN OF MEDIANS?

O(n) work outside of recursive calls

(base case, set-up within MEDIAN_OF_MEDIANS, partitioning)

T(n/5) work hidden in this recursive call

(remember, MEDIAN_OF_MEDIANS calls SELECT on ⌈n/5⌉-size array)

T(???) work hidden in this recursive call

What is the maximum size of either L or R?

ANALYZING RUNTIME

MEDIAN_OF_MEDIANS will choose a pivot greater than at least 3n/10 - 6 elements

(The same reasoning we're about to do also shows that the pivot will be less than at least 3n/10 - 6 elements)

$m = \lceil n/5 \rceil$ groups

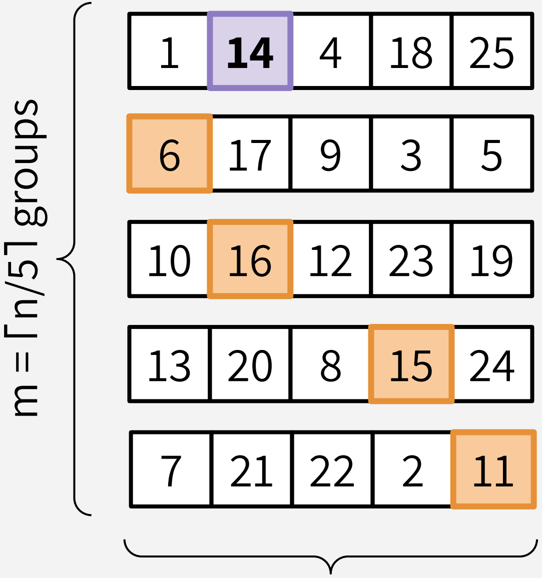
1	14	4	18	25
6	17	9	3	5
10	16	12	23	19
13	20	8	15	24
7	21	22	2	11

at most 5 elements

ANALYZING RUNTIME

MEDIAN_OF_MEDIANS will choose a pivot greater than at least 3n/10 - 6 elements

(The same reasoning we’re about to do also shows that the pivot will be less than at least 3n/10 - 6 elements)



1	**14**	4	18	25
6	17	9	3	5
10	16	12	23	19
13	20	8	15	24
7	21	22	2	11

m = ⌈n/5⌉ groups

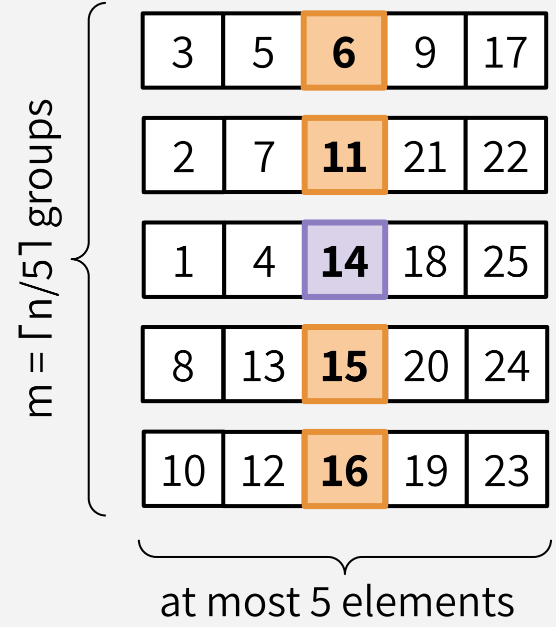
at most 5 elements

At least how many elements are guaranteed to be **smaller** than the median of medians?

ANALYZING RUNTIME

MEDIAN_OF_MEDIANS will choose a pivot greater than at least 3n/10 - 6 elements

(The same reasoning we're about to do also shows that the pivot will be less than at least 3n/10 - 6 elements)



3	5	**6**	9	17
2	7	**11**	21	22
1	4	**14**	18	25
8	13	**15**	20	24
10	12	**16**	19	23

m = ⌈n/5⌉ groups

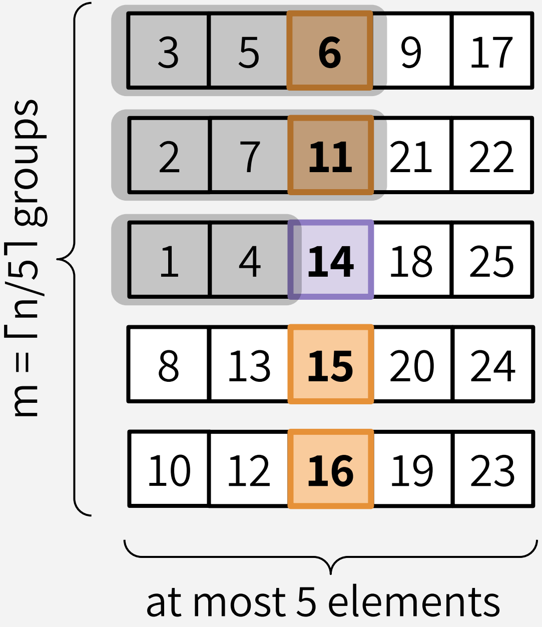
at most 5 elements

At least how many elements are guaranteed to be **smaller** than the median of medians?

ANALYZING RUNTIME

MEDIAN_OF_MEDIANS will choose a pivot greater than at least 3n/10 - 6 elements

(The same reasoning we're about to do also shows that the pivot will be less than at least 3n/10 - 6 elements)



3	5	**6**	9	17
2	7	**11**	21	22
1	4	**14**	18	25
8	13	**15**	20	24
10	12	**16**	19	23

At least how many elements are guaranteed to be **smaller** than the median of medians?

3 elements from each group that has a **median** smaller than the **median of medians** + 2 elements from the group containing the **median of medians**

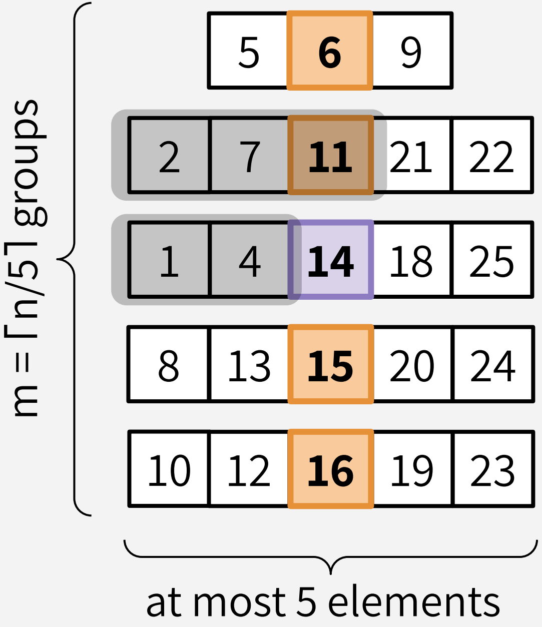
$$3 \cdot (\lceil m/2 \rceil - 1) + 2$$

To exclude the group with the **median of medians**

ANALYZING RUNTIME

MEDIAN_OF_MEDIANS will choose a pivot greater than at least 3n/10 - 6 elements

(The same reasoning we're about to do also shows that the pivot will be less than at least 3n/10 - 6 elements)



	5	**6**	9	
2	7	**11**	21	22
1	4	**14**	18	25
8	13	**15**	20	24
10	12	**16**	19	23

At least how many elements are guaranteed to be **smaller** than the median of medians?

3 elements from each (non-leftover) group that has a **median** smaller than the **median of medians** + 2 elements from the group containing the **median of medians**

$$3 \cdot (\lceil m/2 \rceil - 1 - 1) + 2$$

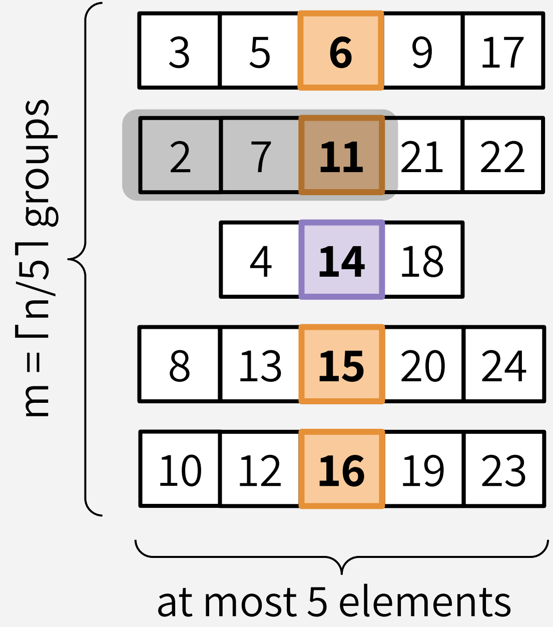
To exclude the group with the **median of medians**

To exclude any of those groups that might be a "leftover" group!

ANALYZING RUNTIME

MEDIAN_OF_MEDIANS will choose a pivot greater than at least 3n/10 - 6 elements

(The same reasoning we're about to do also shows that the pivot will be less than at least 3n/10 - 6 elements)



3	5	**6**	9	17
2	7	**11**	21	22
	4	**14**	18	
8	13	**15**	20	24
10	12	**16**	19	23

At least how many elements are guaranteed to be **smaller** than the median of medians?

3 elements from each (non-leftover) group that has a **median** smaller than the **median of medians** + ~~2 elements from the group containing the **median of medians**~~

$$3 \cdot (\lceil m/2 \rceil - 1 - 1) \cancel{+ 2}$$

To exclude the group with the **median of medians**

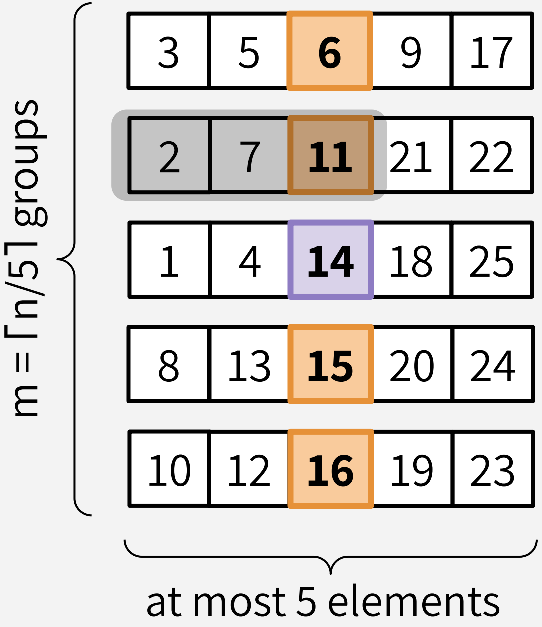
To exclude any of those groups that might be a "leftover" group!

The group with the **median of medians** might be a "leftover" group! Might as well just get rid of the +2 to be safe

ANALYZING RUNTIME

MEDIAN_OF_MEDIANS will choose a pivot greater than at least 3n/10 - 6 elements

(The same reasoning we're about to do also shows that the pivot will be less than at least 3n/10 - 6 elements)



3	5	**6**	9	17
2	7	**11**	21	22
1	4	**14**	18	25
8	13	**15**	20	24
10	12	**16**	19	23

At least how many elements are guaranteed to be **smaller** than the median of medians?

3 elements from each (non-leftover) group that has a **median** smaller than the **median of medians**

$$
\begin{aligned}
& 3 \cdot (\lceil m/2 \rceil - 2) \\
= \; & 3 \cdot (\lceil \lceil n/5 \rceil / 2 \rceil - 2) \\
\geq \; & 3 \cdot (n/10 - 2) \\
= \; & 3n/10 - 6
\end{aligned}
$$

ANALYZING RUNTIME

We just showed:

$$3n/10 - 6 \le \texttt{len(L)}$$

$$\texttt{len(R)} \le 7n/10 + 5$$

ANALYZING RUNTIME

We can similarly show the inverse:

$$3n/10 - 6 \leq \texttt{len(L)} \leq 7n/10 + 5$$

$$3n/10 - 6 \leq \texttt{len(R)} \leq 7n/10 + 5$$

ANALYZING RUNTIME

```
SELECT(A,k):
    if len(A) == 1:
        return A[0]
    p = MEDIAN_OF_MEDIANS(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
        return p
    else if len(L) > k-1:
        return SELECT(L, k)
    else:
        return SELECT(R, k-len(L)-1)
```

O(n) work outside of recursive calls

(base case, set-up within MEDIAN_OF_MEDIANS, partitioning)

T(n/5) work hidden in this recursive call

(remember, MEDIAN_OF_MEDIANS calls SELECT on $\lceil n/5 \rceil$-size array)

T(???) work hidden in this recursive call

What is the maximum size of either L or R?

ANALYZING RUNTIME

We can similarly show the inverse:

$$3n/10 - 6 \leq \texttt{len(L)} \leq 7n/10 + 5$$

$$3n/10 - 6 \leq \texttt{len(R)} \leq 7n/10 + 5$$

What does the recurrence relation for T(n) look like?

$$T(n) \leq T(n/5) + T(???) + O(n)$$

ANALYZING RUNTIME

We can similarly show the inverse:

$$3n/10 - 6 \leq \texttt{len(L)} \leq 7n/10 + 5$$

$$3n/10 - 6 \leq \texttt{len(R)} \leq 7n/10 + 5$$

What does the recurrence relation for T(n) look like?

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

ANALYZING RUNTIME

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

Can be solved by Substitution Method!

SUBSTITUTION METHOD

$T(n) = T(n/5) + T(7n/10) + n$

$T(n) = 1$ when $1 \le n \le 10$

→

Our guess:

T(n) is O(n)

Proof:

We can choose C = 10!

- **Inductive Hypothesis**: $T(n) \le \mathbf{10}n$
- **Base case**: Prove IH holds for $1 \le n \le 10$. $T(n) = 1 \le \mathbf{10}n$
- **Inductive step**:
  - Let $k > 10$. Assume that the IH holds for all n such that $1 \le n < k$.
  - $$\begin{aligned} T(k) &= k + T(k/5) + T(7k/10) \\ &\le k + \mathbf{10} \cdot (k/5) + \mathbf{10} \cdot (7k/10) \\ &= k + 2k + 7k \\ &= \mathbf{10}k \end{aligned}$$
  - Thus, the IH holds for $n = k$!
- **Conclusion:** With $C = 10$ and $n_0 = 1$, $T(n) \le Cn$ for all $n \ge n_0$. By the Big-O definition, $T(n) = O(n)$.

ANALYZING RUNTIME

$$T(n) \le T(n/5) + T(7n/10) + O(n)$$

Can be solved by Substitution Method!

⬇

$$O(n)$$

Worst-case Runtime!

LINEAR-TIME SELECTION

```
SELECT(A,k):
    if len(A) == 1:
        return A[0]
    p = MEDIAN_OF_MEDIANS(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
        return p
    else if len(L) > k-1:
        return SELECT(L, k)
    else if len(L) < k-1:
        return SELECT(R, k-len(L)-1)
```

O(n)

Worst-case Runtime!

سوال؟