Problem solving.

How do you sort a set of objects?
Problem solving.

How do you sort a set of objects?
In high school.

Before you came to Berkeley.

How would you sort by yourself?
Talk to your neighbor!

How would you sort with two of you?
Do some work splitting up?
Do some work putting together?
Talk to your neighbor.
Problem solving.

How do you sort a set of objects?

In high school. Before you came to Berkeley.
Problem solving.

How do you sort a set of objects?

In high school. Before you came to Berkeley.
How do you sort a set of objects?

In high school. Before you came to Berkeley.

How would you sort by yourself?
How do you sort a set of objects?
In high school. Before you came to Berkeley.
How would you sort by yourself?
Talk to your neighbor!
Problem solving.

How do you sort a set of objects?
   In high school. Before you came to Berkeley.
How would you sort by yourself?
   Talk to your neighbor!
How would you sort with two of you?
Problem solving.

How do you sort a set of objects?

In high school. Before you came to Berkeley.

How would you sort by yourself?

Talk to your neighbor!

How would you sort with two of you?
Problem solving.

How do you sort a set of objects?
   In high school. Before you came to Berkeley.
How would you sort by yourself?
   Talk to your neighbor!
How would you sort with two of you?
   Do some work splitting up?
Problem solving.

How do you sort a set of objects?
   In high school. Before you came to Berkeley.
How would you sort by yourself?
   Talk to your neighbor!
How would you sort with two of you?
   Do some work splitting up?
How do you sort a set of objects?
   In high school. Before you came to Berkeley.
How would you sort by yourself?
   Talk to your neighbor!
How would you sort with two of you?
   Do some work splitting up?
   Do some work putting together?
Problem solving.

How do you sort a set of objects?
   In high school. Before you came to Berkeley.
How would you sort by yourself?
   Talk to your neighbor!
How would you sort with two of you?
   Do some work splitting up?
   Do some work putting together?
Talk to your neighbor.
Pick the largest one,
set it aside,
sort the rest,
and put the largest last.
By yourself: $O(n^2)$

Pick the largest one,
set it aside,
sort the rest,
and put the largest last.

Choose the largest one: $O(n)$. 
By yourself: $O(n^2)$

Pick the largest one,  
set it aside,  
sort the rest,  
and put the largest last.

Choose the largest one: $O(n)$.

Do this $n$ times.
By yourself: \( O(n^2) \)

Pick the largest one, set it aside, sort the rest, and put the largest last.

Choose the largest one: \( O(n) \).

Do this \( n \) times.

\( O(n^2) \).
Two of you: divide and conquer.

Split the objects in half.
Two of you: divide and conquer.

Split the objects in half.
Each of you sort.
Two of you: divide and conquer.

Split the objects in half.

Each of you sort.

Merge the sorted piles.
Two of you: divide and conquer.

Split the objects in half.
Each of you sort.
Merge the sorted piles.
  Divide and conquer.
Two of you: divide and conquer.

Split the objects into a larger and smaller group.
Two of you: divide and conquer.

Split the objects into a larger and smaller group. One sorts the larger one, the other the small one.
Two of you: divide and conquer.

Split the objects into a larger and smaller group.
One sorts the larger one, the other the small one.
Put together.
Two of you: divide and conquer.

Split the objects into a larger and smaller group.
One sorts the larger one, the other the small one.
Put together.
  Divide and conquer.
What’s the point of all this?

You are already problem solvers!
What’s the point of all this?

You are already problem solvers!
We are giving tools to be more precise ...
What’s the point of all this?

You are already problem solvers!
We are giving tools to be more precise ... and powerful (with recursion and programming and algorithms.)
More divide and conquer: mergesort.

Sort items in $n$ elt array: $A = [a_1, \ldots, a_n]$,
More divide and conquer: mergesort.

Sort items in $n$ elt array: $A = [a_1, \ldots, a_n]$, E.g., $A = [5, 6, 7, 9, 10, 2, 3\ldots]$. 

```plaintext
Mergesort(A)
if (length(A) >1)
return (merge(mergesort(a[1],...,a[n/2]),
mergesort(a[n/2+1],...,a[n])))
else
return a
```
More divide and conquer: mergesort.

Sort items in \( n \) elt array: \( A = [a_1, \ldots, a_n] \),
E.g., \( A = [5, 6, 7, 9, 10, 2, 3 \ldots] \).

**Mergesort(A)**

```bash
if (length(A) > 1)
  return
    (merge(mergesort(a[1], \ldots, a[n/2]),
        mergesort(a[n/2+1], \ldots, a[n]))
else
  return a
```
More divide and conquer: mergesort.

Sort items in \( n \) elt array: \( A = [a_1, \ldots, a_n] \),
E.g., \( A = [5, 6, 7, 9, 10, 2, 3\ldots] \).

**Mergesort(A)**

\[
\text{if (length(A) >1) return (merge(mergesort(a[1],\ldots,a[n/2]),}
\]\[
\text{mergesort(a[n/2+1],\ldots,a[n]))}
\]\[
\text{else}
\]\[
\text{return a}
\]

How to merge?
More divide and conquer: mergesort.

Sort items in \( n \) elt array: \( A = [a_1, \ldots, a_n] \),
E.g., \( A = [5, 6, 7, 9, 10, 2, 3 \ldots] \).

**Mergesort(A)**

```c
if (length(A) > 1)
    return
    (merge(mergesort(a[1], \ldots, a[n/2]),
           mergesort(a[n/2+1], \ldots, a[n]))
else
    return a
```

How to merge?

Choose lowest from two lists,
More divide and conquer: mergesort.

Sort items in $n$ elt array: $A = [a_1, \ldots, a_n]$, E.g., $A = [5, 6, 7, 9, 10, 2, 3\ldots]$.

**Mergesort(A)**

```python
if (length(A) > 1)
    return
    (merge(mergesort(a[1], \ldots, a[n/2]),
            mergesort(a[n/2+1], \ldots, a[n]))
else
    return a
```

How to merge?

Choose lowest from two lists, cross out,
More divide and conquer: mergesort.

Sort items in \( n \) elt array: \( A = [a_1, \ldots, a_n] \),
E.g., \( A = [5, 6, 7, 9, 10, 2, 3\ldots] \).

\[
\text{Mergesort}(A)
\]
\[
\begin{align*}
\text{if } (\text{length}(A) &> 1) \\
\quad &\text{return}
\quad (\text{merge(mergesort}(a[1], \ldots, a[n/2]),} \\
\quad &\quad \text{mergesort}(a[n/2+1], \ldots, a[n]))
\end{align*}
\]
\[
\text{else}
\quad \text{return } a
\]

How to merge?

Choose lowest from two lists, cross out, repeat.
More divide and conquer: mergesort.

Sort items in \( n \) elt array: \( A = [a_1, \ldots, a_n] \),
E.g., \( A = [5, 6, 7, 9, 10, 2, 3\ldots] \).

**Mergesort(A)**

if (length(A) > 1)
    return
    (merge(mergesort(a[1], \ldots, a[n/2]),
           mergesort(a[n/2+1], \ldots, a[n]))
else
    return a

How to merge?

Choose lowest from two lists, cross out, repeat.

Sorted SubArray 1: 3, 7, 8, 10, 11,\ldots
Sorted Subarray 2: 4, 5, 9, 19, 20,\ldots
More divide and conquer: mergesort.

Sort items in $n$ elt array: $A = [a_1, \ldots, a_n]$, 
E.g., $A = [5, 6, 7, 9, 10, 2, 3\ldots]$.

**Mergesort(A)**

```java
if (length(A) > 1)
    return
    (merge(mergesort(a[1], \ldots, a[n/2]),
           mergesort(a[n/2+1], \ldots, a[n]))
else
    return a
```

How to merge?

Choose lowest from two lists, cross out, repeat.

Sorted SubArray 1: $\mathbf{x}$, 7, 8, 10, 11,\ldots

Sorted Subarray 2: 4, 5, 9, 19, 20,\ldots

3, , , ,
More divide and conquer: mergesort.

Sort items in $n$ elt array: $A = [a_1, \ldots, a_n]$, E.g., $A = [5, 6, 7, 9, 10, 2, 3\ldots]$.

**Mergesort(A)**

```plaintext
if (length(A) >1) return
   (merge(mergesort(a[1],...,a[n/2]),
         mergesort(a[n/2+1],...,a[n]))
else return a
```

How to merge?

Choose lowest from two lists, cross out, repeat.

Sorted SubArray 1: X, 7, 8, 10, 11, \ldots
Sorted Subarray 2: X, 5, 9, 19, 20, \ldots
3, 4, , ,
More divide and conquer: mergesort.

Sort items in $n$ elt array: $A = [a_1, \ldots, a_n]$, 
E.g., $A = [5, 6, 7, 9, 10, 2, 3\ldots]$.

**Mergesort(A)**

```plaintext
if (length(A) > 1)
    return
    (merge(mergesort(a[1], \ldots, a[n/2]),
           mergesort(a[n/2+1], \ldots, a[n]))
else
    return a
```

How to merge?

Choose lowest from two lists, cross out, repeat.

| Sorted SubArray 1: | 3, 7, 8, 10, 11, \ldots |
| Sorted Subarray 2: | 4, 5, 9, 19, 20, \ldots |

3, 4, 5, \ldots
More divide and conquer: mergesort.

Sort items in $n$ elt array: $A = [a_1, \ldots, a_n]$

E.g., $A = [5, 6, 7, 9, 10, 2, 3\ldots]$.

**Mergesort(A)**

```plaintext
if (length(A) > 1)
    return
    (merge(mergesort(a[1], \ldots, a[n/2]),
            mergesort(a[n/2+1], \ldots, a[n])))
else
    return a
```

How to merge?

Choose lowest from two lists, cross out, repeat.

Sorted SubArray 1: $\underline{3}, \underline{X}, 8, 10, 11, \ldots$

Sorted Subarray 2: $\underline{X}, \underline{5}, 9, 19, 20, \ldots$

$3, 4, 5, 7,$
More divide and conquer: mergesort.

Sort items in \( n \) elt array: \( A = [a_1, \ldots, a_n] \),
E.g., \( A = [5, 6, 7, 9, 10, 2, 3\ldots] \).

**Mergesort(A)**

\[
\text{if (length(A) > 1)} \\
\quad \text{return} \\
\quad (\text{merge(mergesort(a[1], \ldots, a[n/2]),} \\
\quad \quad \text{mergesort(a[n/2+1], \ldots, a[n]))})
\]

\[
\text{else} \\
\quad \text{return a}
\]

How to merge?

Choose lowest from two lists, cross out, repeat.

Sorted SubArray 1: 3, 7, 8, 10, 11, ...

Sorted Subarray 2: 4, 5, 9, 19, 20, ...

3, 4, 5, 7, 8
More divide and conquer: mergesort.

Sort items in $n$ elt array: $A = [a_1, \ldots, a_n]$.
E.g., $A = [5, 6, 7, 9, 10, 2, 3\ldots]$.

**Mergesort(A)**

if (length(A) > 1)
    return
    (merge(mergesort(a[1], \ldots, a[n/2]),
            mergesort(a[n/2+1], \ldots, a[n]))
else
    return a

How to merge?

Choose lowest from two lists, cross out, repeat.

Sorted SubArray 1: $\times, \times, \times, 10, 11, \ldots$
Sorted Subarray 2: $\times, \times, 9, 19, 20, \ldots$
3, 4, 5, 7, 8 \ldots
Mergesort: running time analysis

**Mergesort(A)**

if (length(A) > 1)
    return
    (merge(mergesort( a[1],...a[n/2]),
            mergesort( a[n/2+1],...,a[n]))
else
    return a

Split: $O(n)$ time
Could be $O(1)$, e.g., MergeSort(A,start,finish).

Merge: each element in output takes one comparison:
$O(n)$.

Recursive: 2 subproblems of size $n/2$.

$T(n) = 2T(n/2) + O(n)$.

Masters:
$T(n) = aT(n/b) + O(nd)$ with $\log b/a = d = \Rightarrow O(n d \log b n)$.

Apply Masters:
$a = 2, b = 2, d = 1 = \Rightarrow \log 2 2 = 1 = \Rightarrow T(n) = O(n \log n)$.
Mergesort: running time analysis

**Mergesort**(*A*)

if (length(*A*) > 1)
    return
    (merge(mergesort( *a*[1],...,*a*[n/2]),
            mergesort( *a*[n/2+1],...,*a*[n])))
else
    return *a*

Split: \(O(n)\) time
Mergesort: running time analysis

**Mergesort(A)**

```python
if (length(A) >1)
    return
    (merge(mergesort( a[1],...,a[n/2]),
        mergesort( a[n/2+1],...,a[n]))
else
    return a
```

**Split: O(n) time**

Could be O(1), e.g., `MergeSort(A,start,finish)`. 
Mergesort: running time analysis

**Mergesort(A)**

```
if (length(A) >1)
    return
    (merge(mergesort( a[1],...,a[n/2]),
             mergesort( a[n/2+1],...,a[n])))
else
    return a
```

Split: $O(n)$ time

Could be $O(1)$, e.g., `MergeSort(A,start,finish)`. Merge:

```
Mergesort: running time analysis

Mergesort(A)
    if (length(A) >1)
        return
        (merge(mergesort( a[1],...,a[n/2]),
            mergesort( a[n/2+1],...,a[n])))
    else
        return a

Split:  \(O(n)\) time
    Could be \(O(1)\), e.g., \texttt{MergeSort(A,start,finish)}.
Merge: each element in output takes one comparison
Mergesort: running time analysis

Mergesort(A)
if (length(A) > 1)
    return
    (merge(mergesort(a[1],...,a[n/2]),
        mergesort(a[n/2+1],...,a[n]))
else
    return a

Split: $O(n)$ time
   Could be $O(1)$, e.g., MergeSort(A,start,finish).
Merge: each element in output takes one comparison: $O(n)$. 
Mergesort: running time analysis

Mergesort(A)
   if (length(A) > 1)
      return
      (merge(mergesort(a[1],...,a[n/2]),
             mergesort(a[n/2+1],...,a[n]))
   else
      return a

Split: \( O(n) \) time
   Could be \( O(1) \), e.g., MergeSort(A,start,finish).
Merge: each element in output takes one comparison : \( O(n) \).
Recursive: 2 subproblems of size \( n/2 \).
Mergesort: running time analysis

Mergesort(A)
  if (length(A) > 1)
    return
    (merge(mergesort( a[1],...,a[n/2]),
            mergesort( a[n/2+1],...,a[n]))
  else
    return a

Split: $O(n)$ time
Could be $O(1)$, e.g., MergeSort(A,start,finish).

Merge: each element in output takes one comparison: $O(n)$.
Recursive: 2 subproblems of size $n/2$.

$T(n) = 2T(n/2) + O(n)$. 
Mergesort: running time analysis

**Mergesort(A)**

```
if (length(A) > 1)
    return
    (merge(mergesort( a[1],...,a[n/2]),
            mergesort( a[n/2+1],...,a[n])))
else
    return a
```

Split: \(O(n)\) time
Could be \(O(1)\), e.g., **MergeSort(A,start,finish)**.

Merge: each element in output takes one comparison : \(O(n)\).

Recursive: 2 subproblems of size \(n/2\).

\[ T(n) = 2T\left(\frac{n}{2}\right) + O(n). \]

Masters: \( T(n) = aT(n/b) + O(n^d) \) with \( \log_b a = d \)
Mergesort: running time analysis

Mergesort(A)
if (length(A) > 1)
    return
    (merge(mergesort(a[1],...,a[n/2]),
            mergesort(a[n/2+1],...,a[n]))
else
    return a

Split: $O(n)$ time
Could be $O(1)$, e.g., MergeSort(A,start,finish).

Merge: each element in output takes one comparison: $O(n)$.

Recursive: 2 subproblems of size $n/2$.

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

Masters: $T(n) = aT(n/b) + O(n^d)$
with $\log_b a = d \implies O(n^d \log_b n)$
Mergesort: running time analysis

**Mergesort(A)**

```
if (length(A) > 1)
    return
    (merge(mergesort( a[1],...,a[n/2]),
               mergesort( a[n/2+1],...,a[n]))
else
    return a
```

Split: \( O(n) \) time

Could be \( O(1) \), e.g., \( \text{MergeSort}(A, \text{start}, \text{finish}) \).

Merge: each element in output takes one comparison: \( O(n) \).

Recursive: 2 subproblems of size \( n/2 \).

\[
T(n) = 2T\left(\frac{n}{2}\right) + O(n).
\]

Masters: \( T(n) = aT(n/b) + O(n^d) \)

\[
\text{with } \log_b a = d \implies O(n^d \log_b n)
\]
Mergesort: running time analysis

Mergesort(A)
    if (length(A) > 1)
        return
            (merge(mergesort(a[1],...,a[n/2]),
                 mergesort(a[n/2+1],...,a[n]))
    else
        return a

Split: \( O(n) \) time
    Could be \( O(1) \), e.g., MergeSort(A,start,finish).
Merge: each element in output takes one comparison : \( O(n) \).
Recursive: 2 subproblems of size \( n/2 \).

\[
T(n) = 2T\left(\frac{n}{2}\right) + O(n).
\]

Masters: \( T(n) = aT(n/b) + O(n^d) \)
    with \( \log_b a = d \) \( \implies \) \( O(n^d \log_b n) \)

Apply Masters:
\( a = 2, b = 2, d = 1 \)
Mergesort: running time analysis

Mergesort(A)
    if (length(A) >1)
        return
        (merge(mergesort( a[1],...,a[n/2]),
                mergesort( a[n/2+1],...,a[n])))
    else
        return a

Split: $O(n)$ time
Could be $O(1)$, e.g., MergeSort(A,start,finish).

Merge: each element in output takes one comparison : $O(n)$.

Recursive: 2 subproblems of size $n/2$.

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

Masters: $T(n) = aT(n/b) + O(n^d)$
    with $\log_b a = d \implies O(n^d \log_b n)$

Apply Masters:
$a = 2$, $b = 2$, $d = 1 \implies \log_2 2 = 1$
Mergesort: running time analysis

Mergesort(A)
  if (length(A) > 1)
    return
    (merge(mergesort( a[1],...,a[n/2]),
            mergesort( a[n/2+1],...,a[n]))
  else
    return a

Split: $O(n)$ time
  Could be $O(1)$, e.g., MergeSort(A,start,finish).
Merge: each element in output takes one comparision: $O(n)$.  
Recursive: 2 subproblems of size $n/2$.

\[
T(n) = 2T\left(\frac{n}{2}\right) + O(n).
\]

Masters: $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$
  with $\log_b a = d \implies O(n^d \log_b n)$

Apply Masters:
  $a = 2, b = 2, d = 1 \implies \log_2 2 = 1 \implies T(n) = O(n \log n)$. 
Iterative Mergesort: Bottom up, use of queues.

Make each element into list and put lists in queue.

Merge first two lists, put in queue (at end).

Rinse.

Repeat.

Each pass through queue:
each element touched once.

$O(n)$ time.

Each pass halves number of lists.

$= \Rightarrow O(\log n)$ passes

$= \Rightarrow O(n \log n)$ time
Iterative Mergesort: Bottom up, use of queues.
Make each element into list and put lists in queue.
Check it out...

Iterative Mergesort: Bottom up, use of queues.
Make each element into list and put lists in queue.

\[
\begin{array}{cccc}
8 & 3 & 5 & 9 \\
\end{array}
\]

\cdots \cdots
Check it out...

Iterative Mergesort: Bottom up, use of queues.
Make each element into list and put lists in queue.

8 3 5 9  

Merge first two lists, put in queue (at end).

5 9 3,8
Check it out...

Iterative Mergesort: Bottom up, use of queues.
Make each element into list and put lists in queue.

```
8  3  5  9  ······
```

Merge first two lists, put in queue (at end).

```
5  9  ······  3,8
```

Rinse.
Iterative Mergesort: Bottom up, use of queues.

Make each element into list and put lists in queue.

| 8 | 3 | 5 | 9 |   |

Merge first two lists, put in queue (at end).

| 5 | 9 |   |

Rinse. Repeat.
Check it out...

Iterative Mergesort: Bottom up, use of queues.

Make each element into list and put lists in queue.

\[
\begin{array}{cccccc}
8 & 3 & 5 & 9 & \cdots & \cdots \\
\end{array}
\]

Merge first two lists, put in queue (at end).

\[
\begin{array}{ccccccc}
5 & 9 & \cdots & \cdots & 3,8 \\
\end{array}
\]

Rinse. Repeat.

\[
\begin{array}{cccc}
\cdots & \cdots & 3,8 & 5,9 \\
\end{array}
\]
Iterative Mergesort: Bottom up, use of queues.

Make each element into list and put lists in queue.

```
8 3 5 9     ······
```

Merge first two lists, put in queue (at end).

```
5 9     ······  3,8
```

Rinse. Repeat.

```
······  3,8  5,9
```

And next pass through queue...
Check it out...

Iterative Mergesort: Bottom up, use of queues.

Make each element into list and put lists in queue.

\[
\begin{matrix}
8 & 3 & 5 & 9 & \cdots & \cdots \\
\end{matrix}
\]

Merge first two lists, put in queue (at end).

\[
\begin{matrix}
5 & 9 & \cdots & \cdots & 3,8 \\
\end{matrix}
\]

Rinse. Repeat.

\[
\begin{matrix}
\cdots & \cdots & 3,8 & 5,9 \\
\end{matrix}
\]

And next pass through queue...

\[
\begin{matrix}
\cdots & \cdots & 3,5,8,9 \\
\end{matrix}
\]

Each pass through queue:
- Each element touched once. \(O(n)\) time.
- Each pass halves number of lists. \(\Rightarrow O(\log n)\) passes = \(O(n \log n)\) time.
Check it out...

Iterative Mergesort: Bottom up, use of queues.

Make each element into list and put lists in queue.

| 8 | 3 | 5 | 9 | ... | ...

Merge first two lists, put in queue (at end).

| 5 | 9 | ... | ... | 3,8 |

Rinse. Repeat.

| ... | ... | 3,8 | 5,9 |

And next pass through queue...

| ... | ... | 3,5,8,9 |

Each pass through queue:
Check it out...

Iterative Mergesort: Bottom up, use of queues.

Make each element into list and put lists in queue.

```plaintext
8 3 5 9  · · · · ·
```

Merge first two lists, put in queue (at end).

```plaintext
5 9  · · · · · 3,8
```

Rinse. Repeat.

```plaintext
· · · · · 3,8 5,9
```

And next pass through queue...

```plaintext
· · · · · 3,5,8,9
```

Each pass through queue: each element touched once.
Check it out...

Iterative Mergesort: Bottom up, use of queues.
Make each element into list and put lists in queue.

\[
\begin{array}{cccc}
8 & 3 & 5 & 9 \\
\end{array}
\]

Merge first two lists, put in queue (at end).

\[
\begin{array}{cccc}
5 & 9 & & \\
\end{array}
\]

Rinse. Repeat.

\[
\begin{array}{cccc}
 & & 3,8 \\
3,8 & 5,9 & & \\
\end{array}
\]

And next pass through queue...

\[
\begin{array}{cccc}
 & & 3,5,8,9 \\
3,5,8,9 & & & \\
\end{array}
\]

Each pass through queue: each element touched once. \(O(n)\) time.
Iterative Mergesort: Bottom up, use of queues.

Make each element into list and put lists in queue.

| 8 | 3 | 5 | 9 | ····· |

Merge first two lists, put in queue (at end).

| 5 | 9 | ····· | 3,8 |

Rinse. Repeat.

| ····· | 3,8 | 5,9 |

And next pass through queue...

| ····· | 3,5,8,9 |

Each pass through queue: each element touched once. $O(n)$ time. Each pass halves number of lists.
Iterative Mergesort: Bottom up, use of queues.

Make each element into list and put lists in queue.

| 8 | 3 | 5 | 9 |   |   |   |

Merge first two lists, put in queue (at end).

| 5 | 9 |   |   |   |   | 3, 8 |

Rinse. Repeat.

|   |   |   | 3, 8 |   | 5, 9 |

And next pass through queue...

|   |   |   | 3, 5, 8, 9 |

Each pass through queue: each element touched once. $O(n)$ time.

Each pass halves number of lists.

$\Rightarrow O(\log n)$ passes
Check it out...

Iterative Mergesort: Bottom up, use of queues.
Make each element into list and put lists in queue.

\[
\begin{array}{cccc}
8 & 3 & 5 & 9 \\
\end{array}
\]

Merge first two lists, put in queue (at end).

\[
\begin{array}{cccc}
5 & 9 & \cdot & \cdot \\
\end{array}
\]

Rinse. Repeat.

\[
\begin{array}{cccc}
\cdot & \cdot & \cdot & \cdot \\
\end{array}
\]

And next pass through queue...

\[
\begin{array}{cccc}
\cdot & \cdot & \cdot & \cdot \\
3, 5, 8, 9
\end{array}
\]

Each pass through queue: each element touched once. \(O(n)\) time.
Each pass halves number of lists.

\[\Rightarrow O(\log n) \text{ passes} \Rightarrow O(n \log n) \text{ time}\]
Sorting lower bound.

Can we do better?
Can we do better?

Comparison sorting algorithm only compares numbers.
Can we do better?

Comparison sorting algorithm only compares numbers. Does not look at bits only uses result of comparison.
Can we do better?

Comparison sorting algorithm only compares numbers. Does not look at bits only uses result of comparison.

Merge:
Compare two first elts and then output first.
Can we do better?

Comparison sorting algorithm only compares numbers. Does not look at bits only uses result of comparison.

Merge:
Compare two first elts and then output first.

Comparison sort?
Can we do better?

Comparison sorting algorithm only compares numbers. Does not look at bits only uses result of comparison.

Merge:
Compare two first elts and then output first.

Comparison sort? Yes.
Can we do better?

Comparison sorting algorithm only compares numbers. Does not look at bits only uses result of comparison.

Merge:
Compare two first elts and then output first.

Comparison sort? Yes.

“Radix” Sort.
Sorting lower bound.

Can we do better?

Comparison sorting algorithm only compares numbers. Does not look at bits only uses result of comparison.

Merge:
Compare two first els and then output first.

Comparison sort? Yes.

“Radix” Sort.
Bucket according to whether begins with “A”, “B”....
Can we do better?

Comparison sorting algorithm only compares numbers. Does not look at bits only uses result of comparison.

Merge:
Compare two first elts and then output first.

Comparison sort? Yes.

“Radix” Sort.
Bucket according to whether begins with “A”, “B”.... Repeat in each bucket with next characters.
Can we do better?
Comparison sorting algorithm only compares numbers. Does not look at bits only uses result of comparison.

Merge:
Compare two first elts and then output first.

Comparison sort? Yes.

“Radix” Sort.
Bucket according to whether begins with “A”, “B”.... Repeat in each bucket with next characters.
Looks at characters...
Can we do better?

Comparison sorting algorithm only compares numbers. Does not look at bits only uses result of comparison.

Merge:
Compare two first elts and then output first.

Comparison sort? Yes.

“Radix” Sort.
Bucket according to whether begins with “A”, “B”.... Repeat in each bucket with next characters.
Looks at characters... or looks at “bits”.

Can we do better?

Comparison sorting algorithm only compares numbers. Does not look at bits only uses result of comparison.

Merge:
Compare two first elts and then output first.

Comparison sort? Yes.

“Radix” Sort.
Bucket according to whether begins with “A”, “B”.... Repeat in each bucket with next characters. Looks at characters... or looks at “bits”.
Not a comparison sort!
Thm: Comparison sort requires $\Omega(n \log n)$ comparisons.
Thm: Comparison sort requires $\Omega(n \log n)$ comparisons.

**Proof idea:** Input: $a_1, a_2, \ldots, a_n$
Thm: Comparison sort requires $\Omega(n \log n)$ comparisons.

Proof idea: Input: $a_1, a_2, \ldots, a_n$
Possible Output: $a_8, a_{n-8}, \ldots, a_{15}$
Thm: Comparison sort requires $\Omega(n \log n)$ comparisons.

**Proof idea:**

Input: $a_1, a_2, \ldots, a_n$

Possible Output: $a_8, a_{n-8}, \ldots, a_{15}$

Represent output as permutation of $[1, \ldots, n]$. 

Sorting lower bound.
Thm: Comparison sort requires $\Omega(n \log n)$ comparisons.

Proof idea: Input: $a_1, a_2, \ldots, a_n$
Possible Output: $a_8, a_{n-8}, \ldots, a_{15}$
Represent output as permutation of $[1, \ldots, n]$.
Output: $8, n-8, \ldots, 15$. 
Thm: Comparison sort requires $\Omega(n \log n)$ comparisons.

Proof idea: Input: $a_1, a_2, \ldots, a_n$
Possible Output: $a_8, a_{n-8}, \ldots, a_{15}$
Represent output as permutation of $[1, \ldots, n]$.
Output: $8, n-8, \ldots, 15$.

How many possible outputs?
Thm: Comparison sort requires $\Omega(n \log n)$ comparisons.

**Proof idea:** Input: $a_1, a_2, \ldots, a_n$
Possible Output: $a_8, a_{n-8}, \ldots, a_{15}$
Represent output as permutation of $[1, \ldots, n]$.
Output: $8, n-8, \ldots, 15$.

How many possible outputs? $n!$
Thm: Comparison sort requires $\Omega(n \log n)$ comparisons.

**Proof idea:** Input: $a_1, a_2, \ldots, a_n$
Possible Output: $a_8, a_{n-8}, \ldots, a_{15}$
Represent output as permutation of $[1, \ldots, n]$.
Output: $8, n-8, \ldots, 15$.

How many possible outputs? $n!$

Algorithm must be about to output any of $n!$ permutations.
Thm: Comparison sort requires $\Omega(n \log n)$ comparisons.

Proof idea: Input: $a_1, a_2, \ldots, a_n$
Possible Output: $a_8, a_{n-8}, \ldots, a_{15}$
Represent output as permutation of $[1, \ldots, n]$.
Output: $8, n-8, \ldots, 15$.

How many possible outputs? $n!$
Algorithm must be about to output any of $n!$ permutations.
Algorithm must output just 1 permutation at termination.
Algorithm must be able to output any of $n!$ permutations.
Algorithm must be able to output any of $n!$ permutations. Algorithm must output just 1 permutation at termination.
Algorithm must be able to output any of $n!$ permutations.
Algorithm must output just 1 permutation at termination.
Algorithm must be able to output any of $n!$ permutations. Algorithm must output just 1 permutation at termination. Algorithm as tree of comparisons.
Algorithm must be able to output any of $n!$ permutations.
Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons.
After a sequence of comparisons get to termination or 1 permutation.
Algorithm must be able to output any of $n!$ permutations. Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons. After a sequence of comparisons get to termination or 1 permutation.

$S$ is set of possible permutations at some point in Algorithm
Algorithm must be able to output any of $n!$ permutations. Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons.
After a sequence of comparisons get to termination or 1 permutation.

$S$ is set of possible permutations at some point in Algorithm

Example:
Algorithm must be able to output any of $n!$ permutations. Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons.
   After a sequence of comparisons get to termination or 1 permutation.

$S$ is set of possible permutations at some point in Algorithm.
   Example: After no comparisons, any output is possible.
Algorithm must be able to output any of $n!$ permutations. Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons. After a sequence of comparisons get to termination or 1 permutation.

$S$ is set of possible permutations at some point in Algorithm

Example: After no comparisons, any output is possible.
Algorithm must be able to output any of $n!$ permutations. Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons. After a sequence of comparisons get to termination or 1 permutation.

$S$ is set of possible permutations at some point in Algorithm

Example: After no comparisons, any output is possible.

Do some comparison: $a_i > a_j$?
Algorithm must be able to output any of \( n! \) permutations. 
Algorithm must output just 1 permutation at termination. 

Algorithm as tree of comparisons. 
After a sequence of comparisons get to termination or 1 permutation.

\( S \) is set of possible permutations at some point in Algorithm
Example: After no comparisons, any output is possible.

Do some comparison: \( a_i > a_j \)?
Algorithm must be able to output any of $n!$ permutations.
Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons.
After a sequence of comparisons get to termination or 1 permutation.

$S$ is set of possible permutations at some point in Algorithm
Example: After no comparisons, any output is possible.

Do some comparison: $a_i > a_j$?
If Yes, Alg “could” return subset of permutations: $S_1$. 

---

$S_1 \cup S_2 = S \Rightarrow \max(|S_1|, |S_2|) \geq |S|/2.$
Each comparison divides possible outputs by at most 2.
Need at least $\log_2(n!)$ comparisons to get to just 1 permutation.

$n! \geq (ne)^n = \Rightarrow \log n! = \Omega(n \log n)$.
Algorithm must be able to output any of \( n! \) permutations. Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons.
After a sequence of comparisons get to termination or 1 permutation.

\( S \) is set of possible permutations at some point in Algorithm
Example: After no comparisons, any output is possible.

Do some comparision: \( a_i > a_j \)?

If Yes, Alg “could” return subset of permutations: \( S_1 \).
If No, Alg “could” return subset of permutations: \( S_2 \).
Algorithm must be able to output any of $n!$ permutations.
Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons.
  After a sequence of comparisons get to termination or 1 permutation.

$S$ is set of possible permutations at some point in Algorithm
  Example: After no comparisons, any output is possible.

Do some comparison: $a_i > a_j$?
  If Yes, Alg “could” return subset of permutations: $S_1$.
  If No, Alg “could” return subset of permutations: $S_2$.

$S_1 \cup S_2 = S$
Algorithm must be able to output any of $n!$ permutations. Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons. After a sequence of comparisons get to termination or 1 permutation.

$S$ is set of possible permutations at some point in Algorithm

Example: After no comparisons, any output is possible.

Do some comparison: $a_i > a_j$?

If Yes, Alg “could” return subset of permutations: $S_1$. If No, Alg “could” return subset of permutations: $S_2$.

$S_1 \cup S_2 = S \implies \max(|S_1|, |S_2|) \geq |S|/2$. 

Each comparison divides possible outputs by at most 2. Need at least $\log_2(n!)$ comparisons to get to just 1 permutation.

$n! \geq (ne)^n \implies \log(n!) = \Omega(n \log n)$. 

Sorting lower bound: ...proof
Algorithm must be able to output any of \( n! \) permutations. Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons.

After a sequence of comparisons get to termination or 1 permutation.

\( S \) is set of possible permutations at some point in Algorithm

Example: After no comparisons, any output is possible.

Do some comparison: \( a_i > a_j? \)

If Yes, Alg “could” return subset of permutations: \( S_1 \).
If No, Alg “could” return subset of permutations: \( S_2 \).

\( S_1 \cup S_2 = S \iff \max(|S_1|, |S_2|) \geq |S|/2. \)

Each comparison divides possible outputs by at most 2.
Algorithm must be able to output any of $n!$ permutations. Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons. After a sequence of comparisons get to termination or 1 permutation.

$S$ is set of possible permutations at some point in Algorithm

Example: After no comparisons, any output is possible.

Do some comparison: $a_i > a_j$?

If Yes, Alg “could” return subset of permutations: $S_1$.
If No, Alg “could” return subset of permutations: $S_2$.

$S_1 \cup S_2 = S \implies \max(|S_1|, |S_2|) \geq |S|/2$.

Each comparison divides possible outputs by at most 2.

Need at least $\log_2(n!)$ comparisons to get to just 1 permutation.
Algorithm must be able to output any of \(n!\) permutations.
Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons.
After a sequence of comparisons get to termination or 1 permutation.

\(S\) is set of possible permutations at some point in Algorithm

Example: After no comparisons, any output is possible.

Do some comparision: \(a_i > a_j\)?

If Yes, Alg “could” return subset of permutations: \(S_1\).
If No, Alg “could” return subset of permutations: \(S_2\).

\(S_1 \cup S_2 = S \implies \max(|S_1|, |S_2|) \geq |S|/2.\)

Each comparision divides possible outputs by at most 2.

Need at least \(\log_2(n!)\) comparisions to get to just 1 permutation.
...to get to termination.
Algorithm must be able to output any of \( n! \) permutations.
Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons.
After a sequence of comparisons get to termination or 1 permutation.

\( S \) is set of possible permutations at some point in Algorithm

Example: After no comparisons, any output is possible.

Do some comparison: \( a_i > a_j \)?

If Yes, Alg “could” return subset of permutations: \( S_1 \).
If No, Alg “could” return subset of permutations: \( S_2 \).

\[ S_1 \cup S_2 = S \implies \max(|S_1|, |S_2|) \geq |S|/2. \]
Each comparison divides possible outputs by at most 2.

Need at least \( \log_2(n!) \) comparisons to get to just 1 permutation.
...to get to termination.

\[ n! \geq \left( \frac{n}{e} \right)^n \]
Sorting lower bound: ...proof

Algorithm must be able to output any of $n!$ permutations. Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons.
   After a sequence of comparisons get to termination or 1 permutation.

$S$ is set of possible permutations at some point in Algorithm
   Example: After no comparisons, any output is possible.

Do some comparision: $a_i > a_j$?
   If Yes, Alg “could” return subset of permutations: $S_1$.
   If No, Alg “could” return subset of permutations: $S_2$.

$S_1 \cup S_2 = S \implies \max(|S_1|, |S_2|) \geq |S|/2$.

Each comparision divides possible outputs by at most 2.

Need at least $\log_2(n!)$ comparisions to get to just 1 permutation.
   ...to get to termination.

$n! \geq (\frac{n}{e})^n \implies \log n!$
Algorithm must be able to output any of $n!$ permutations.  
Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons. 
After a sequence of comparisons get to termination or 1 permutation.

$S$ is set of possible permutations at some point in Algorithm
Example: After no comparisons, any output is possible.

Do some comparison: $a_i > a_j$?

If Yes, Alg “could” return subset of permutations: $S_1$.
If No, Alg “could” return subset of permutations: $S_2$.

$S_1 \cup S_2 = S \implies \max(|S_1|, |S_2|) \geq |S|/2$.

Each comparison divides possible outputs by at most 2.

Need at least $\log_2(n!)$ comparisons to get to just 1 permutation.
...to get to termination.

\[ n! \geq \left(\frac{n}{e}\right)^n \implies \log n! = \Omega(\log(n^n)) \]
Algorithm must be able to output any of $n!$ permutations. Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons. After a sequence of comparisons get to termination or 1 permutation.

$S$ is set of possible permutations at some point in Algorithm

Example: After no comparisons, any output is possible.

Do some comparison: $a_i > a_j$?

If Yes, Alg “could” return subset of permutations: $S_1$.
If No, Alg “could” return subset of permutations: $S_2$.

$S_1 \cup S_2 = S \implies \max(|S_1|, |S_2|) \geq |S|/2$.

Each comparison divides possible outputs by at most 2.

Need at least $\log_2(n!)$ comparisons to get to just 1 permutation.
...to get to termination.

$n! \geq \left(\frac{n}{e}\right)^n \implies \log n! = \Omega(\log(n^n)) = \Omega(n \log n)$.
Sorting lower bound: ...proof

Algorithm must be able to output any of \( n! \) permutations.
Algorithm must output just 1 permutation at termination.

Algorithm as tree of comparisons.
After a sequence of comparisons get to termination or 1 permutation.

\( S \) is set of possible permutations at some point in Algorithm
Example: After no comparisons, any output is possible.

Do some comparison: \( a_i > a_j \)?

If Yes, Alg “could” return subset of permutations: \( S_1 \).
If No, Alg “could” return subset of permutations: \( S_2 \).

\[ S_1 \cup S_2 = S \implies \max(|S_1|, |S_2|) \geq |S|/2. \]

Each comparison divides possible outputs by at most 2.

Need at least \( \log_2(n!) \) comparisons to get to just 1 permutation.
...to get to termination.

\[ n! \geq \left( \frac{n}{e} \right)^n \implies \log n! = \Omega(\log(n^n)) = \Omega(n \log n). \]
Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half. Depth must be $\Omega(\log(\#\text{permutations})) = \Omega(\log n!) = \Omega(n \log n)$.

Can we do better than mergesort? Yes? No. For comparison sort. (Recall from 61b: radix sort may be faster: $O(n)$.)

A research area: "bit complexity" versus "word complexity".
Either the set of permutations $S_1$ or $S_2$ is larger.
Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half.
Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half. Depth must be $\Omega(\log(\#\text{permutations}))$.
Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half. Depth must be $\Omega(\log(\#\text{permutations})) = \Omega(\log n!)$
Figure for proof.

Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half.
Depth must be $\Omega(\log(\#\text{permutations})) = \Omega(\log n!) = \Omega(n \log n)$. 
Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half. Depth must be $\Omega(\log(\#\text{permutations})) = \Omega(\log n!) = \Omega(n \log n)$.

Can we do better than mergesort?
Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half. Depth must be $\Omega(\log(\#\text{permutations})) = \Omega(\log n!) = \Omega(n \log n)$.

Can we do better than mergesort? Yes?
Figure for proof.

Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half.

Depth must be $\Omega(\log(\# \text{permutations})) = \Omega(\log n!) = \Omega(n \log n)$.

Can we do better than mergesort? Yes? No?
Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half. Depth must be $\Omega(\log(\#\text{permutations})) = \Omega(\log n!) = \Omega(n \log n)$.

Can we do better than mergesort? Yes? No?
Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half.

Depth must be $\Omega(\log(\#\text{permutations})) = \Omega(\log n!) = \Omega(n \log n)$.

Can we do better than mergesort? Yes? No? No.
Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half. Depth must be $\Omega(\log(\#\text{permutations})) = \Omega(\log n!) = \Omega(n \log n)$.

Can we do better than mergesort? Yes? No?

No. For comparison sort.

(Recall from 61b: radix sort may be faster: $O(n)$.)

A research area: "bit complexity" versus "word complexity".
Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half.

Depth must be $\Omega(\log(\#\text{permutations})) = \Omega(\log n!) = \Omega(n\log n)$.

Can we do better than mergesort? Yes? No?

No. For comparison sort.

(Recall from 61b: radix sort may be faster: $O(n)$.)
Either the set of permutations $S_1$ or $S_2$ is larger. One must be at least half.

Depth must be $\Omega(\log(\# \text{permutations})) = \Omega(\log n!) = \Omega(n \log n)$.

Can we do better than mergesort? Yes? No?

No. For comparison sort.

(Recall from 61b: radix sort may be faster: $O(n)$.)

A research area: “bit complexity” versus “word complexity”.
Radix sort and lower bound.

Why is radix sort not subject to lower bound?
Radix sort and lower bound.

Why is radix sort not subject to lower bound?
It buckets by 'letter'.
Radix sort and lower bound.

Why is radix sort not subject to lower bound? It buckets by 'letter'. So, the degree of the tree is not 2.
Radix sort and lower bound.

Why is radix sort not subject to lower bound?
It buckets by 'letter'.
So, the degree of the tree is not 2. It is larger.
Median finding.

Find the median element of a set of elements: \( a_1, \ldots, a_n \).
Median finding.

Find the median element of a set of elements: $a_1, \ldots, a_n$.
Median is value, $v$, where $\frac{n}{2}$ elts are less than $v$ (if $n$ is odd.)
Median finding.

Find the median element of a set of elements: \( a_1, \ldots, a_n \).

Median is value, \( v \), where \( \frac{n}{2} \) elts are less than \( v \) (if \( n \) is odd.)

Versus Average?

Average household income (2004): $70,700

Median household income (2004): $43,200

Why so different?

Bill Gates and Jeff Bezos.

Why use average?

Find average?

Compute \( \sum_{i=1}^{n} a_i \).

\( O(n) \) time.

Compute median?

Sort to get \( s_1, \ldots, s_n \).

Output element \( s_{n/2 + 1} \).

\( O(n \log n) \) time.

Better algorithm?
Median finding.

Find the median element of a set of elements: $a_1,\ldots,a_n$.
Median is value, $v$, where $\frac{n}{2}$ elts are less than $v$ (if $n$ is odd.)

Versus Average?

Average household income (2004): $70,700
Median finding.

Find the median element of a set of elements: $a_1, \ldots, a_n$.
Median is value, $v$, where $\frac{n}{2}$ elts are less than $v$ (if $n$ is odd.)

Versus Average?

Average household income (2004): $70,700
Median household income (2004): $43,200
Median finding.

Find the median element of a set of elements: \( a_1, \ldots, a_n \).
Median is value, \( v \), where \( \frac{n}{2} \) elts are less than \( v \) (if \( n \) is odd.)

Versus Average?

Average household income (2004): $70,700
Median household income (2004): $43,200
Median finding.

Find the median element of a set of elements: $a_1, \ldots, a_n$.
Median is value, $v$, where $\frac{n}{2}$ elts are less than $v$ (if $n$ is odd.)

Versus Average?

Average household income (2004): $70,700
Median household income (2004): $43,200

Why so different?
Find the median element of a set of elements: $a_1, \ldots, a_n$.
Median is value, $v$, where $\frac{n}{2}$ elts are less than $v$ (if $n$ is odd.)

Versus Average?

Average household income (2004): $70,700
Median household income (2004): $43,200

Why so different? Bill Gates and Jeff Bezos.
Median finding.

Find the median element of a set of elements: $a_1, \ldots, a_n$.

Median is value, $v$, where $\frac{n}{2}$ elts are less than $v$ (if $n$ is odd.)

Versus Average?

Average household income (2004): $70,700
Median household income (2004): $43,200

Why so different? Bill Gates and Jeff Bezos. The 1%, perhaps.
Median finding.

Find the median element of a set of elements: $a_1, \ldots, a_n$.
Median is value, $v$, where $\frac{n}{2}$ elts are less than $v$ (if $n$ is odd.)

Versus Average?

Average household income (2004): $70,700
Median household income (2004): $43,200

Why so different? Bill Gates and Jeff Bezos. The 1%, perhaps.

Why use average?
Median finding.

Find the median element of a set of elements: \( a_1, \ldots, a_n \).
Median is value, \( v \), where \( \frac{n}{2} \) elts are less than \( v \) (if \( n \) is odd.)

Versus Average?
Average household income (2004): $70,700
Median household income (2004): $43,200
Why so different? Bill Gates and Jeff Bezos. The 1%, perhaps.

Why use average?
Find average?
Median finding.

Find the median element of a set of elements: \( a_1, \ldots, a_n \).
Median is value, \( v \), where \( \frac{n}{2} \) elts are less than \( v \) (if \( n \) is odd.)

Versus Average?

Average household income (2004): $70,700
Median household income (2004): $43,200

Why so different? Bill Gates and Jeff Bezos. The 1%, perhaps.

Why use average?

Find average? Compute \( \frac{\sum_i a_i}{n} \).
Median finding.

Find the median element of a set of elements: \( a_1, \ldots, a_n \).
Median is value, \( v \), where \( \frac{n}{2} \) elts are less than \( v \) (if \( n \) is odd.)

Versus Average?
Average household income (2004): $70,700
Median household income (2004): $43,200

Why so different? Bill Gates and Jeff Bezos. The 1%, perhaps.

Why use average?
Find average? Compute \( \frac{\sum_i a_i}{n} \).
\( O(n) \) time.
Median finding.

Find the median element of a set of elements: $a_1, \ldots, a_n$.
Median is value, $v$, where $\frac{n}{2}$ elts are less than $v$ (if $n$ is odd.)

Versus Average?

Average household income (2004): $70,700
Median household income (2004): $43,200

Why so different? Bill Gates and Jeff Bezos. The 1%, perhaps.

Why use average?

Find average? Compute $\frac{\sum_i a_i}{n}$.
$O(n)$ time.

Compute median?
Median finding.

Find the median element of a set of elements: \( a_1, \ldots, a_n \).

Median is value, \( v \), where \( \frac{n}{2} \) elts are less than \( v \) (if \( n \) is odd.)

Versus Average?

Average household income (2004): $70,700
Median household income (2004): $43,200

Why so different? Bill Gates and Jeff Bezos. The 1%, perhaps.

Why use average?

Find average? Compute \( \frac{\sum a_i}{n} \).

\( O(n) \) time.

Compute median? Sort to get \( s_1, \ldots s_n \).
Median finding.

Find the median element of a set of elements: $a_1, \ldots, a_n$.
Median is value, $v$, where $\frac{n}{2}$ elts are less than $v$ (if $n$ is odd.)

Versus Average?

Average household income (2004): $70,700
Median household income (2004): $43,200

Why so different? Bill Gates and Jeff Bezos. The 1%, perhaps.

Why use average?

Find average? Compute $\frac{\sum_i a_i}{n}$.
$O(n)$ time.

Compute median? Sort to get $s_1, \ldots s_n$. Output element $s_{n/2+1}$. 
Median finding.

Find the median element of a set of elements: \(a_1, \ldots, a_n\).
Median is value, \(v\), where \(\frac{n}{2}\) elts are less than \(v\) (if \(n\) is odd.)

Versus Average?

Average household income (2004): $70,700
Median household income (2004): $43,200

Why so different? Bill Gates and Jeff Bezos. The 1%, perhaps.

Why use average?

Find average? Compute \(\frac{\sum_i a_i}{n}\).
\(O(n)\) time.

Compute median? Sort to get \(s_1, \ldots s_n\). Output element \(s_{n/2+1}\).
\(O(n \log n)\) time.
Median finding.

Find the median element of a set of elements: $a_1, \ldots, a_n$.
Median is value, $\nu$, where $\frac{n}{2}$ elts are less than $\nu$ (if $n$ is odd.)

Versus Average?

Average household income (2004): $70,700
Median household income (2004): $43,200

Why so different? Bill Gates and Jeff Bezos. The 1%, perhaps.

Why use average?

Find average? Compute $\frac{\sum_i a_i}{n}$.
$O(n)$ time.

Compute median? Sort to get $s_1, \ldots s_n$. Output element $s_{n/2+1}$.
$O(n \log n)$ time.

Better algorithm?
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \), select the \( k \)th smallest element.

Median: select \( \lfloor n/2 \rfloor + 1 \) elt.

Example. \( k = 7 \) for items \{11, 48, 5, 21, 2, 15, 17, 19, 15\}.

Output?

(A) 19
(B) 15
(C) 21
Solve a harder Problem: Selection.

For a set of $n$ items $S$. 

Example. $k = 7$ for items $\{11, 48, 5, 21, 2, 15, 17, 19, 15\}$ 

Output? 

(A) 19 
(B) 15 
(C) 21
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.
Median: select $\lfloor n/2 \rfloor + 1$ elt.
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lfloor n/2 \rfloor + 1$ elt.

Example.
$k = 7$ for items $\{11, 48, 5, 21, 2, 15, 17, 19, 15\}$
Solve a harder Problem: Selection.

For a set of $n$ items $S$. 
Select $k$th smallest element.

Median: select $\lceil n/2 \rceil + 1$ elt.

Example.
$k = 7$ for items $\{11, 48, 5, 21, 2, 15, 17, 19, 15\}$
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lfloor n/2 \rfloor + 1$ elt.

Example.
$k = 7$ for items $\{11, 48, 5, 21, 2, 15, 17, 19, 15\}$

Output?

(A) 19
(B) 15
(C) 21
Solve a harder Problem: Selection.

For a set of $n$ items $S$. Select $k$th smallest element. 

Median: select $\lfloor n/2 \rfloor + 1$ elt.

Example. 
$k = 7$ for items $\{11, 48, 5, 21, 2, 15, 17, 19, 15\}$ 

Output? 

(A) 19 
(B) 15 
(C) 21
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \).
Select \( k \)th smallest element.

Median: select \( \lfloor n/2 \rfloor + 1 \) elt.

Example.
\( k = 7 \) for items \( \{11, 48, 5, 21, 2, 15, 17, 19, 15\} \)

Output?

(A) 19
(B) 15
(C) 21

???
Solve a harder Problem: Selection.

For a set of $n$ items $S$, select $k$th smallest element.

Median: select $\lfloor n/2 \rfloor + 1$ elt.

$$(k, S)$$:

$\begin{cases} k = 7 & S : 11, 48, 5, 21, 2, 15, 17, 19, 15 \\ \text{Base Case: } k = 1 \text{ and } |S| = 1, \text{ return elt.} \\ \text{Choose rand. pivot elt } b \text{ from } A. \\ v = 15 \\ \text{Form } S_L \text{ containing all elts } < v \\ S_L : 11, 5, 2 \\ \text{Form } S_v \text{ containing all elts } = v \\ S_v : 15, 15 \\ \text{Form } S_R \text{ containing all elts } > v \\ S_R : 48, 21, 17, 19 \\ \text{If } k \leq |S_L|, \text{ Select } (k, S_L). \\ \text{elseif } k \leq |S_L| + |S_v|, \text{ return } v. \\ \text{else Select } (k - |S_L| - |S_v|, S_R) \end{cases}$$

Will eventually return 19, which is 7th element of list.

Correctness: Induction. Idea: Subroutine returns correct answer, and so will I! Base case is good. Subroutine calls ..by design.
Solve a harder Problem: Selection.

For a set of $n$ items $S$. 
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lfloor n/2 \rfloor + 1$ elt.
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \).
Select \( k \)th smallest element.

Median: select \( \lceil n/2 \rceil + 1 \) elt.

**Select**\( (k, S) \): \( k = 7 \) \( \quad S : 11, 48, 5, 21, 2, 15, 17, 19, 15 \)
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lfloor n/2 \rfloor + 1$ elt.

**Select**($k, S$): $k = 7$

Base Case: $k = 1$ and $|S| = 1$, return elt.
Choose rand. pivot elt $b$ from $A$.

$S : 11, 48, 5, 21, 2, 15, 17, 19, 15$

$v = 15$

Will eventually return 19, which is 7th element of list.
Correctness:
Induction.
Idea: Subroutine returns correct answer, and so will $I$!
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lfloor n/2 \rfloor + 1$ elt.

$\text{Select}(k, S)$: $k = 7$  
$S : 11, 48, 5, 21, 2, 15, 17, 19, 15$
Base Case: $k = 1$ and $|S| = 1$, return elt.
Choose rand. pivot elt $b$ from $A$.
Form $S_L$ containing all elts $< v$

Will eventually return 19, which is 7th element of list.

Correctness:
Induction.
Idea: Subroutine returns correct answer, and so will $I$!
Base case is good.
Subroutine calls ..by design.
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lceil n/2 \rceil + 1$ elt.

**Select**($k, S$): $k = 7$  
$S : 11, 48, 5, 21, 2, 15, 17, 19, 15$
Base Case: $k = 1$ and $|S| = 1$, return elt.
Choose rand. pivot elt $b$ from $A$.
Form $S_L$ containing all elts $< v$
$v = 15$
$S_L : 11, 5, 2$

Will eventually return 19, which is 7th element of list.

Correctness: Induction.
Idea: Subroutine returns correct answer, and so will I!
Base case is good.
Subroutine calls ..by design.
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lceil n/2 \rceil + 1$ elt.

**Select**($k$, $S$): $k = 7$

Base Case: $k = 1$ and $|S| = 1$, return elt.
Choose rand. pivot elt $b$ from $A$.
Form $S_L$ containing all elts $< v$
Form $S_v$ containing all elts $= v$

$S : 11, 48, 5, 21, 2, 15, 17, 19, 15$
$v = 15$
$S_L : 11, 5, 2$

Will eventually return 19, which is 7th element of list.

Correctness: Induction.
Idea: Subroutine returns correct answer, and so will I!
Base case is good.
Subroutine calls ..by design.
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lceil n/2 \rceil + 1$ elt.

Select($k, S$): $k = 7$

Base Case: $k = 1$ and $|S| = 1$, return elt.
Choose rand. pivot elt $b$ from $A$.
Form $S_L$ containing all elts $< v$
Form $S_V$ containing all elts $= v$

$S : 11, 48, 5, 21, 2, 15, 17, 19, 15$
$v = 15$
$S_L : 11, 5, 2$
$S_V : 15, 15$

Will eventually return 19, which is 7th element of list.

Correctness:
Induction.
Idea: Subroutine returns correct answer, and so will $I$!
Base case is good.
Subroutine calls..by design.
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lceil n/2 \rceil + 1$ elt.

**Select**$(k, S)$: $k = 7$

<table>
<thead>
<tr>
<th>$S$</th>
<th>$S_L$</th>
<th>$S_V$</th>
<th>$S_R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>11, 48, 5, 21, 2, 15, 17, 19, 15</td>
<td>11, 5, 2</td>
<td>15, 15</td>
<td>48, 21, 17, 19</td>
</tr>
</tbody>
</table>

Base Case: $k = 1$ and $|S| = 1$, return elt.
Choose rand. pivot elt $b$ from $A$.
Form $S_L$ containing all elts $< v$
Form $S_V$ containing all elts $= v$
Form $S_R$ containing all elts $> v$

$\nu = 15$
$S_L = 11, 5, 2$
$S_V = 15, 15$

Will eventually return 19, which is 7th element of list.

Correctness: Induction.
Idea: Subroutine returns correct answer, and so will I! 
Base case is good. Subroutine calls by design.
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \).
Select \( k \)th smallest element.

Median: select \( \lceil n/2 \rceil + 1 \) elt.

**Select**\((k, S)\): \( k = 7 \)

Base Case: \( k = 1 \) and \( |S| = 1 \), return elt.
Choose rand. pivot elt \( b \) from \( A \).
Form \( S_L \) containing all elts \(< v \)
Form \( S_V \) containing all elts \( = v \)
Form \( S_R \) containing all elts \( > v \)

\( S : 11, 48, 5, 21, 2, 15, 17, 19, 15 \)
\( v = 15 \)
\( S_L : 11, 5, 2 \)
\( S_V : 15, 15 \)
\( S_R : 48, 21, 17, 19 \)

Will eventually return 19, which is 7th element of list.

Correctness:
Induction.
Idea: Subroutine returns correct answer, and so will I!
Base case is good.
Subroutine calls ..by design.
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lceil n/2 \rceil + 1$ elt.

Select($k, S$): $k = 7$  
$S : 11, 48, 5, 21, 2, 15, 17, 19, 15$

Base Case: $k = 1$ and $|S| = 1$, return elt.
Choose rand. pivot elt $b$ from $A$.

Form $S_L$ containing all elts $< v$
Form $S_v$ containing all elts $= v$
Form $S_R$ containing all elts $> v$

If $k \leq |S_L|$, Select($k, S_L$).

$v = 15$

$S_L : 11, 5, 2$
$S_v : 15, 15$
$S_R : 48, 21, 17, 19$

$7 \leq 3?$
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lceil n/2 \rceil + 1$ elt.

**Select**($k, S$): $k = 7$

Base Case: $k = 1$ and $|S| = 1$, return elt.
Choose rand. pivot elt $b$ from $A$.
Form $S_L$ containing all elts $< v$
Form $S_V$ containing all elts $= v$
Form $S_R$ containing all elts $> v$

If $k \leq |S_L|$, Select($k, S_L$).
elseif $k \leq |S_L| + |S_V|$, return $v$.

\[ S : 11, 48, 5, 21, 2, 15, 17, 19, 15 \]
\[ v = 15 \]
\[ S_L : 11, 5, 2 \]
\[ S_V : 15, 15 \]
\[ S_R : 48, 21, 17, 19 \]

Will eventually return 19, which is 7th element of list.
Correctness: Induction.
Idea: Subroutine returns correct answer, and so will $I$!
Base case is good.
Subroutine calls ..by design.
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \).
Select \( k \)th smallest element.

Median: select \( \lceil n/2 \rceil + 1 \) elt.

**Select**\((k, S)\): \( k = 7 \)  
Base Case: \( k = 1 \) and \( |S| = 1 \), return elt.
Choose rand. pivot elt \( b \) from \( A \).
Form \( S_L \) containing all elts < \( v \)
Form \( S_V \) containing all elts = \( v \)
Form \( S_R \) containing all elts > \( v \)

If \( k \leq |S_L| \), Select\((k, S_L)\).
elseif \( k \leq |S_L| + |S_V| \), return \( v \).
else Select\((k - |S_L| - |S_V|, S_R)\)
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \).
Select \( k \)th smallest element.

Median: select \( \lfloor n/2 \rfloor + 1 \) elt.

**Select**\((k, S)\): \( k = 7 \)

Base Case: \( k = 1 \) and \( |S| = 1 \), return elt.

Choose rand. pivot elt \( b \) from \( A \).

Form \( S_L \) containing all elts \(< v \)

Form \( S_v \) containing all elts \( = v \)

Form \( S_R \) containing all elts \( > v \)

If \( k \leq |S_L| \), Select\((k, S_L)\).
elseif \( k \leq |S_L| + |S_v| \), return \( v \).
else Select\((k − |S_L| − |S_v|, S_R)\)

\( S : 11, 48, 5, 21, 2, 15, 17, 19, 15 \)

\( v = 15 \)

\( S_L : 11, 5, 2 \)

\( S_v : 15, 15 \)

\( S_R : 48, 21, 17, 19 \)

7 ≤ 3?

7 ≤ 5?

Select\((2, [48, 21, 17, 19])\)
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \).

Select \( k \)th smallest element.

Median: select \( \lfloor n/2 \rfloor + 1 \) elt.

**Select** \((k, S)\): \( k = 7 \)

- **Base Case:** \( k = 1 \) and \( |S| = 1 \), return elt.
- Choose rand. pivot elt \( b \) from \( A \).
- Form \( S_L \) containing all elts \(< v \)
- Form \( S_v \) containing all elts \( = v \)
- Form \( S_R \) containing all elts \( > v \)

- If \( k \leq |S_L| \), Select\((k, S_L)\).
- elseif \( k \leq |S_L| + |S_v| \), return \( v \).
- else Select\((k − |S_L| − |S_v|, S_R)\)

Will eventually return 19, which is 7th element of list.
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \).
Select \( k \)th smallest element.

Median: select \( \lceil n/2 \rceil + 1 \) elt.

**Select**\((k, S)\): \( k = 7 \)

Base Case: \( k = 1 \) and \( |S| = 1 \), return elt.

Choose rand. pivot elt \( b \) from \( A \).

Form \( S_L \) containing all elts < \( b \)
Form \( S_v \) containing all elts = \( b \)
Form \( S_R \) containing all elts > \( b \)

If \( k \leq |S_L| \), Select\((k, S_L)\). \( 7 \leq 3? \)
elseif \( k \leq |S_L| + |S_v| \), return \( b \). \( 7 \leq 5? \)
else Select\((k - |S_L| - |S_v|, S_R)\)

Will eventually return 19, which is 7th element of list.

Correctness:
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lfloor n/2 \rfloor + 1$ elt.

**Select**$(k,S)$: \[ k = 7 \]

Base Case: $k = 1$ and $|S| = 1$, return elt.

Choose rand. pivot elt $b$ from $A$.

Form $S_L$ containing all elts $< v$

Form $S_V$ containing all elts $= v$

Form $S_R$ containing all elts $> v$

If $k \leq |S_L|$, Select($k,S_L$).
elseif $k \leq |S_L| + |S_V|$, return $v$.
else Select($k − |S_L| − |S_V|, S_R$)

Will eventually return 19, which is 7th element of list.

Correctness: Induction.

Idea: Subroutine returns correct answer,
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lceil n/2 \rceil + 1$ elt.

Select$(k, S)$: $k = 7$

Base Case: $k = 1$ and $|S| = 1$, return elt.

Choose rand. pivot elt $b$ from $A$.

Form $S_L$ containing all elts $< v$
Form $S_V$ containing all elts $= v$
Form $S_R$ containing all elts $> v$

If $k \leq |S_L|$, Select$(k, S_L)$.
elseif $k \leq |S_L| + |S_V|$, return $v$.
else Select$(k - |S_L| - |S_V|, S_R)$

Will eventually return 19, which is 7th element of list.

Correctness: Induction.

Idea: Subroutine returns correct answer, and
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \).
Select \( k \)th smallest element.

Median: select \( \lceil n/2 \rceil + 1 \) elt.

**Select** \((k, S)\): \( k = 7 \) \( S : 11, 48, 5, 21, 2, 15, 17, 19, 15 \)
Base Case: \( k = 1 \) and \( |S| = 1 \), return elt.
Choose rand. pivot elt \( b \) from \( A \).
Form \( S_L \) containing all elts \( < v \)
Form \( S_v \) containing all elts \( = v \)
Form \( S_R \) containing all elts \( > v \)

If \( k \leq |S_L| \), Select\((k, S_L)\).
elseif \( k \leq |S_L| + |S_v| \), return \( v \).
else Select\((k − |S_L| − |S_v|, S_R)\)

Will eventually return 19, which is 7th element of list.

Correctness: Induction.
Idea: Subroutine returns correct answer, and so
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \).
Select \( k \)th smallest element.

Median: select \( \lceil n/2 \rceil + 1 \) elt.

**Select**(\( k, S \)):

- \( k = 7 \)
- \( S : 11, 48, 5, 21, 2, 15, 17, 19, 15 \)
- \( v = 15 \)
- \( S_L : 11, 5, 2 \)
- \( S_V : 15, 15 \)
- \( S_R : 48, 21, 17, 19 \)

Base Case: \( k = 1 \) and \( |S| = 1 \), return elt.
Choose rand. pivot elt \( b \) from \( A \).
Form \( S_L \) containing all elts \(< v \)
Form \( S_V \) containing all elts \( = v \)
Form \( S_R \) containing all elts \( > v \)

If \( k \leq |S_L| \), Select\( (k, S_L) \).
elseif \( k \leq |S_L| + |S_V| \), return \( v \).
else Select\( (k − |S_L| − |S_V|, S_R) \)

Will eventually return 19, which is 7th element of list.

Correctness: Induction.
Idea: Subroutine returns correct answer, and so will
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \).
Select \( k \)th smallest element.

Median: select \( \lceil n/2 \rceil + 1 \) elt.

**Select** \((k, S)\): \( k = 7 \)

Base Case: \( k = 1 \) and \(|S| = 1\), return elt.
Choose rand. pivot elt \( b \) from \( A \).
Form \( S_L \) containing all elts \(< v \)
Form \( S_V \) containing all elts \( = v \)
Form \( S_R \) containing all elts \( > v \)

If \( k \leq |S_L| \), Select\((k, S_L)\).
elseif \( k \leq |S_L| + |S_V| \), return \( v \).
else Select\((k - |S_L| - |S_V|, S_R)\)

Will eventually return 19, which is 7th element of list.

Correctness: Induction.
Idea: Subroutine returns correct answer, and so will I
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \).
Select \( k \)th smallest element.

Median: select \( \lceil n/2 \rceil + 1 \) elt.

**Select**\((k,S)\): \( k = 7 \) \( S : 11, 48, 5, 21, 2, 15, 17, 19, 15 \)

Base Case: \( k = 1 \) and \( |S| = 1 \), return elt. 

Choose rand. pivot elt \( b \) from \( A \).

Form \( S_L \) containing all elts < \( \nu \) 

Form \( S_V \) containing all elts = \( \nu \) 

Form \( S_R \) containing all elts > \( \nu \)  

If \( k \leq |S_L| \), Select\((k,S_L)\).  
elseif \( k \leq |S_L| + |S_V| \), return \( \nu \). 
else Select\((k − |S_L| − |S_V|, S_R)\)

Will eventually return 19, which is 7th element of list.

Correctness: Induction. 
Idea: Subroutine returns correct answer, and so will I!
Solve a harder Problem: Selection.

For a set of \( n \) items \( S \).
Select \( k \)th smallest element.

Median: select \( \lfloor n/2 \rfloor + 1 \) elt.

**Select**\((k, S)\): \( k = 7 \)

Base Case: \( k = 1 \) and \( |S| = 1 \), return elt.

Choose rand. pivot elt \( b \) from \( A \).

Form \( S_L \) containing all elts \(< v \)

Form \( S_v \) containing all elts \(= v \)

Form \( S_R \) containing all elts \(> v \)

If \( k \leq |S_L| \), Select\((k, S_L)\).
elseif \( k \leq |S_L| + |S_v| \), return \( v \).
else Select\((k − |S_L| − |S_v|, S_R)\)

Will eventually return 19, which is 7th element of list.

Correctness: Induction.
Idea: Subroutine returns correct answer, and so will I!
Base case is good.
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lceil n/2 \rceil + 1$ elt.

Select($k, S$): $k = 7$

Base Case: $k = 1$ and $|S| = 1$, return elt.

Choose rand. pivot elt $b$ from $A$.
Form $S_L$ containing all elts $< v$
Form $S_V$ containing all elts $= v$
Form $S_R$ containing all elts $> v$

If $k \leq |S_L|$, Select($k, S_L$).
elseif $k \leq |S_L| + |S_V|$, return $v$.
else Select($k - |S_L| - |S_V|, S_R$)

Will eventually return 19, which is 7th element of list.

Correctness: Induction.
Idea: Subroutine returns correct answer, and so will I!
Base case is good. Subroutine calls
Solve a harder Problem: Selection.

For a set of $n$ items $S$.
Select $k$th smallest element.

Median: select $\lceil n/2 \rceil + 1$ elt.

**Select**($k, S$): 

$k = 7$ 

Base Case: $k = 1$ and $|S| = 1$, return elt.

Choose rand. pivot elt $b$ from $A$.

Form $S_L$ containing all elts $< v$

Form $S_V$ containing all elts $= v$

Form $S_R$ containing all elts $> v$

If $k \leq |S_L|$, Select($k, S_L$).
elseif $k \leq |S_L| + |S_V|$, return $v$.
else Select($k - |S_L| - |S_V|, S_R$)

$S : 11, 48, 5, 21, 2, 15, 17, 19, 15$
$v = 15$

$S_L : 11, 5, 2$

$S_V : 15, 15$

$S_R : 48, 21, 17, 19$

Will eventually return 19, which is 7th element of list.

Correctness: Induction.

Idea: Subroutine returns correct answer, and so will I!
Base case is good. Subroutine calls ..by design.
The Induction.

Base Case: \( k = 1, |S| = 1 \). Trivial.
The Induction.

Base Case: $k = 1, |S| = 1$. Trivial.

| $S_L$ | $S_v$ | $S_R$ |
The Induction.

Base Case: $k = 1$, $|S| = 1$. Trivial.

| $S_L$ | $S_v$ | $S_R$ |

If $k \leq |S_L|$, Select($k, S_L$)
The Induction.

Base Case: $k = 1, |S| = 1$. Trivial.

<table>
<thead>
<tr>
<th>$S_L$</th>
<th>$S_v$</th>
<th>$S_R$</th>
</tr>
</thead>
</table>

If $k \leq |S_L|$, Select($k, S_L$)
$k$th element in first $|S_L|$ elts.
The Induction.

Base Case: $k = 1, |S| = 1$. Trivial.

<table>
<thead>
<tr>
<th>$S_L$</th>
<th>$S_v$</th>
<th>$S_R$</th>
</tr>
</thead>
</table>

If $k \leq |S_L|$, Select($k, S_L$)
   $k$th element in first $|S_L|$ elts.
The Induction.

Base Case: $k = 1, |S| = 1$. Trivial.

<table>
<thead>
<tr>
<th>$S_L$</th>
<th>$S_v$</th>
<th>$S_R$</th>
</tr>
</thead>
</table>

If $k \leq |S_L|$, Select($k, S_L$)
- $k$th element in first $|S_L|$ elts.
- $k$th elt of $S$ is $k$th elt of $S_L$
The Induction.

Base Case: $k = 1$, $|S| = 1$. Trivial.

<table>
<thead>
<tr>
<th>$S_L$</th>
<th>$S_v$</th>
<th>$S_R$</th>
</tr>
</thead>
</table>

If $k \leq |S_L|$, Select($k, S_L$)
  $k$th element in first $|S_L|$ elts.
  $k$th elt of $S$ is $k$th elt of $S_L$
elseif $k \leq |S_L| + |S_v|$, return $v$,
The Induction.

Base Case: \( k = 1, |S| = 1 \). Trivial.

<table>
<thead>
<tr>
<th>( S_L )</th>
<th>( S_v )</th>
<th>( S_R )</th>
</tr>
</thead>
</table>

If \( k \leq |S_L| \), Select\((k, S_L)\)
  
  \( k \)th element in first \( |S_L| \) elts.
  
  \( k \)th elt of \( S \) is \( k \)th elt of \( S_L \)

elseif \( k \leq |S_L| + |S_v| \), return \( v \),
  
  \( k \in [|S_L|, \ldots, |S_L| + |S_v|] \).
The Induction.

Base Case: $k = 1$, $|S| = 1$. Trivial.

<table>
<thead>
<tr>
<th></th>
<th>$S_L$</th>
<th>$S_v$</th>
<th>$S_R$</th>
</tr>
</thead>
</table>

If $k \leq |S_L|$, Select($k, S_L$)
- $k$th element in first $|S_L|$ els.
- $k$th elt of $S$ is $k$th elt of $S_L$

elseif $k \leq |S_L| + |S_v|$, return $v$, $k \in [|S_L|, \ldots, |S_L| + |S_v|]$. 

else Select($k - |S_L| - |S_v|$, $S_R$)
- $k$th element is in $S_R$
- $k$th elt of $S$ is $k$th elt of $S_L$ after els of $S_L \cup S_v$. 

Correct in all cases.
The Induction.

Base Case: \( k = 1, \ |S| = 1. \) Trivial.

<table>
<thead>
<tr>
<th>( S_L )</th>
<th>( S_v )</th>
<th>( S_R )</th>
</tr>
</thead>
</table>

If \( k \leq |S_L| \), Select\((k, S_L)\)
  
  \( k \)th element in first \( |S_L| \) elts.
  
  \( k \)th elt of \( S \) is \( k \)th elt of \( S_L \)

elseif \( k \leq |S_L| + |S_v| \), return \( v \),

  \( k \in [|S_L|, \ldots, |S_L| + |S_v|] \).

  \( k \)th elt of \( S \) is in \( S_v \), all have value \( v \)
The Induction.

Base Case: $k = 1$, $|S| = 1$. Trivial.

| $S_L$ | $S_v$ | $S_R$ |

If $k \leq |S_L|$, Select($k, S_L$)
- $k$th element in first $|S_L|$ elts.
- $k$th elt of $S$ is $k$th elt of $S_L$

elseif $k \leq |S_L| + |S_v|$, return $v$,
- $k \in [|S_L|, \ldots, |S_L| + |S_v|]$.
- $k$th elt of $S$ is in $S_v$, all have value $v$

else Select($k - |S_L| - |S_v|, S_R$)
The Induction.

Base Case: $k = 1$, $|S| = 1$. Trivial.

| $S_L$ | $S_v$ | $S_R$ |

If $k \leq |S_L|$, Select($k, S_L$)
- $k$th element in first $|S_L|$ elts.
- $k$th elt of $S$ is $k$th elt of $S_L$

elseif $k \leq |S_L| + |S_v|$, return $v$,
- $k \in [|S_L|, \ldots, |S_L| + |S_v|]$.
- $k$th elt of $S$ is in $S_v$, all have value $v$

else Select($k - |S_L| - |S_v|, S_R$)
- $k$th element is in $S_R$ and
- $k$th elt of $S$ is $k - |S_L| - |S_v|$ after elts of $S_L \cup S_v$. 

Correct in all cases.
The Induction.

Base Case: $k = 1$, $|S| = 1$. Trivial.

| $S_L$ | $S_v$ | $S_R$ |

If $k \leq |S_L|$, Select($k$, $S_L$)
   $k$th element in first $|S_L|$ els.
   $k$th elt of $S$ is $k$th elt of $S_L$
elseif $k \leq |S_L| + |S_v|$, return $v$,
   $k \in [|S_L|, \ldots, |S_L| + |S_v|]$.
   $k$th elt of $S$ is in $S_v$, all have value $v$
else Select($k - |S_L| - |S_v|$, $S_R$)
   $k$th element is in $S_R$ and
   $k$th elt of $S$ is $k - |S_L| - |S_v|$ after els of $S_L \cup S_v$.

Correct in all cases.
The Induction.

Base Case: $k = 1, |S| = 1$. Trivial.

<table>
<thead>
<tr>
<th>$S_L$</th>
<th>$S_v$</th>
<th>$S_R$</th>
</tr>
</thead>
</table>

If $k \leq |S_L|$, Select($k, S_L$)
- $k$th element in first $|S_L|$ elts.
- $k$th elt of $S$ is $k$th elt of $S_L$
elseif $k \leq |S_L| + |S_v|$, return $v$,
- $k \in [|S_L|, \ldots, |S_L| + |S_v|]$.
- $k$th elt of $S$ is in $S_v$, all have value $v$
else Select($k - |S_L| - |S_v|, S_R$)
- $k$th element is in $S_R$ and
- $k$th elt of $S$ is $k - |S_L| - |S_v|$ after elts of $S_L \cup S_v$.

Correct in all cases.
Selection: runtime.

Worst case runtime?
Selection: runtime.

Worst case runtime?

(A) $\Theta(n \log n)$

(B) $\Theta(n)$

(C) $\Theta(n^2)$

Let $k = n$. 

Worse than sorting! On average?
Selection: runtime.

Worst case runtime?

(A) $\Theta(n \log n)$
(B) $\Theta(n)$
(C) $\Theta(n^2)$

Let $k = n$.
Partition element is smallest every time.
Selection: runtime.

Worst case runtime?
(A) $\Theta(n \log n)$
(B) $\Theta(n)$
(C) $\Theta(n^2)$

Let $k = n$.
Partition element is smallest every time.
Size of list decrease by 1 in each recursive call.
Selection: runtime.

Worst case runtime?
(A) \( \Theta(n \log n) \)
(B) \( \Theta(n) \)
(C) \( \Theta(n^2) \)

Let \( k = n \).
Partition element is smallest every time.
Size of list decrease by 1 in each recursive call.
Time for partition is \( O(i) \) time when \( i \) elements.
Selection: runtime.

Worst case runtime?
(A) $\Theta(n \log n)$
(B) $\Theta(n)$
(C) $\Theta(n^2)$

Let $k = n$.
Partition element is smallest every time.
Size of list decrease by 1 in each recursive call.
Time for partition is $O(i)$ time when $i$ elements.
$\Theta(n + (n-1) + \cdots + 2 + 1) = \Theta(n^2)$ time. or (C)
Selection: runtime.

Worst case runtime?
(A) $\Theta(n \log n)$
(B) $\Theta(n)$
(C) $\Theta(n^2)$

Let $k = n$.
Partition element is smallest every time.
Size of list decrease by 1 in each recursive call.
Time for partition is $O(i)$ time when $i$ elements.
$\Theta(n + (n-1) + \cdots + 2 + 1) = \Theta(n^2)$ time. or (C)
Worse than sorting!
Selection: runtime.

Worst case runtime?

(A) $\Theta(n \log n)$
(B) $\Theta(n)$
(C) $\Theta(n^2)$

Let $k = n$.
Partition element is smallest every time.
Size of list decrease by 1 in each recursive call.
Time for partition is $O(i)$ time when $i$ elements.
$\Theta(n + (n-1) + \cdots + 2 + 1) = \Theta(n^2)$ time. or (C)
Worse than sorting!
On average?
Average time to get a heads?

Flip a coin, what is average number of tosses to get a heads?
Average time to get a heads?

Flip a coin, what is average number of tosses to get a heads?

(A) two

(B) three
Flip a coin, what is average number of tosses to get a heads?

(A) two
(B) three
(C) Could go forever!
Average time to get a heads?

Flip a coin, what is average number of tosses to get a heads?

(A) two
(B) three
(C) Could go forever!

(A) ..
Average time to get a heads?

Flip a coin, what is average number of tosses to get a heads?
(A) two
(B) three
(C) Could go forever!

(A) ..and (C)
Flip a coin, what is average number of tosses to get a heads?

(A) two
(B) three
(C) Could go forever!

(A) ..and (C) (but not relevant.)
Lemma:
The expected number of coin tosses to get a heads is 2.

Proof:
\[ E[X] = 1 + \frac{1}{2} E[X] \]
\[ \Rightarrow \frac{1}{2} E[X] = 1 \]
\[ E[X] = 2. \]

Probability that a random pivot element in the middle half is \[ \geq \frac{1}{2} \].

Expected time to get a middle element is \[ E[X] \times O(n) = O(n) \].

Pick in the middle half subproblem size is \[ \leq \frac{3}{4} n \].

Expected time recurrence:
\[ T(n) \leq T\left(\frac{3}{4} n\right) + O(n) \]

Masters or just thinking:
\[ (n + \left(\frac{3}{4} n\right) + \left(\frac{3}{4} n\right)^2 + \cdots) = O(n) \]
\[ \Rightarrow T(n) = O(n) \].
Expected (average) Time?

**Lemma:** Expected number of coin tosses to get a heads is 2.
Expected (average) Time?

Lemma: Expected number of coin tosses to get a heads is 2.

Proof: \[ E[X] = 1 + \frac{1}{2} E[X] \]
Expected (average) Time?

**Lemma:** Expected number of coin tosses to get a heads is 2.

**Proof:**

\[ E[X] = 1 + \frac{1}{2} E[X] \]

\[ \Rightarrow \quad \frac{1}{2} E[X] = 1 \]
Expected (average) Time?

**Lemma:** Expected number of coin tosses to get a heads is 2.

**Proof:** $E[X] = 1 + \frac{1}{2}E[X]$  
$\implies \frac{1}{2}E[X] = 1 \implies E[X] = 2.$
Expected (average) Time?

**Lemma:** Expected number of coin tosses to get a heads is 2.

**Proof:** \( E[X] = 1 + \frac{1}{2} E[X] \)
\[ \implies \frac{1}{2} E[X] = 1 \implies E[X] = 2. \]

Probability that random pivot elt in the **middle half** is \( \geq \frac{1}{2} \).
Expected (average) Time?

**Lemma:** Expected number of coin tosses to get a heads is 2.

**Proof:** \( E[X] = 1 + \frac{1}{2}E[X] \)
\[ \implies \frac{1}{2}E[X] = 1 \implies E[X] = 2. \]

Probability that random pivot elt in the middle half is \( \geq \frac{1}{2} \).
Expected (average) Time?

**Lemma:** Expected number of coin tosses to get a heads is 2.

**Proof:**

\[ E[X] = 1 + \frac{1}{2} E[X] \]
\[ \implies \frac{1}{2} E[X] = 1 \implies E[X] = 2. \]

Probability that random pivot elt in the **middle half** is \( \geq \frac{1}{2} \).

Expected time to get a middle element is \( E[X] \times O(n) = O(n) \).
Expected (average) Time?

**Lemma:** Expected number of coin tosses to get a heads is 2.

**Proof:** $E[X] = 1 + \frac{1}{2}E[X]
\implies \frac{1}{2}E[X] = 1 \implies E[X] = 2.$

Probability that random pivot elt in the middle half is $\geq \frac{1}{2}.$

Expected time to get a middle element is $E[X] \times O(n) = O(n).$

Pick in the middle half subproblem size is $\leq \frac{3}{4}n.$
Expected (average) Time?

**Lemma:** Expected number of coin tosses to get a heads is 2.

**Proof:** $E[X] = 1 + \frac{1}{2}E[X]$

$\implies \frac{1}{2}E[X] = 1 \implies E[X] = 2.$

Probability that random pivot elt in the middle half is $\geq \frac{1}{2}$.

Expected time to get a middle element is $E[X] \times O(n) = O(n)$.

Pick in the middle half subproblem size is $\leq \frac{3}{4}n$.

Expected time recurrence:

$T(n) \leq \ldots$
**Expected (average) Time?**

**Lemma:** Expected number of coin tosses to get a heads is 2.

**Proof:** \( E[X] = 1 + \frac{1}{2} E[X] \)
\[
\implies \frac{1}{2} E[X] = 1 \implies E[X] = 2. \]

Probability that random pivot elt in the **middle half** is \( \geq \frac{1}{2} \).

Expected time to get a middle element is \( E[X] \times O(n) = O(n) \).

Pick in the middle half subproblem size is \( \leq \frac{3}{4} n \).

Expected time recurrence:
\[
T(n) \leq T\left(\frac{3}{4} n\right) + \]
Expected (average) Time?

Lemma: Expected number of coin tosses to get a heads is 2.

Proof: \( E[X] = 1 + \frac{1}{2} E[X] \)
\[ \implies \frac{1}{2} E[X] = 1 \implies E[X] = 2. \]

Probability that random pivot elt in the middle half is \( \geq \frac{1}{2} \).

Expected time to get a middle element is \( E[X] \times O(n) = O(n) \).

Pick in the middle half subproblem size is \( \leq \frac{3}{4} n \).

Expected time recurrence:
\[ T(n) \leq T(\frac{3}{4} n) + O(n). \]
Expected (average) Time?

**Lemma:** Expected number of coin tosses to get a heads is 2.

**Proof:**

\[ E[X] = 1 + \frac{1}{2}E[X] \]

\[ \Rightarrow \frac{1}{2}E[X] = 1 \Rightarrow E[X] = 2. \]

Probability that random pivot elt in the **middle half** is \( \geq \frac{1}{2} \).

Expected time to get a middle element is \( E[X] \times O(n) = O(n) \).

Pick in the middle half subproblem size is \( \leq \frac{3}{4}n \).

Expected time recurrence:

\[ T(n) \leq T\left(\frac{3}{4}n\right) + O(n). \]

Masters or just thinking:
Expected (average) Time?

**Lemma:** Expected number of coin tosses to get a heads is 2.

**Proof:**

\[
E[X] = 1 + \frac{1}{2}E[X]
\]

\[
\implies \frac{1}{2}E[X] = 1 \implies E[X] = 2.
\]

Probability that random pivot elt in the **middle half** is \( \geq \frac{1}{2} \).

Expected time to get a middle element is \( E[X] \times O(n) = O(n) \).

Pick in the middle half subproblem size is \( \leq \frac{3}{4}n \).

Expected time recurrence:

\[
T(n) \leq T(\frac{3}{4}n) + O(n).
\]

Masters or just thinking: \( (n + (3/4)n + (3/4)^2n + \cdots = O(n)) \)
Expected (average) Time?

Lemma: Expected number of coin tosses to get a heads is 2.

Proof: $E[X] = 1 + \frac{1}{2}E[X]$  
$\implies \frac{1}{2}E[X] = 1 \implies E[X] = 2.$

Probability that random pivot elt in the middle half is $\geq \frac{1}{2}$.

Expected time to get a middle element is $E[X] \times O(n) = O(n)$.

Pick in the middle half subproblem size is $\leq \frac{3}{4}n$.

Expected time recurrence:

$$T(n) \leq T\left(\frac{3}{4}n\right) + O(n).$$

Masters or just thinking: $(n + (3/4)n + (3/4)^2n + \cdots = O(n))$  
$\implies T(n) = O(n)$.
Other sorting method.

Find median.
Other sorting method.

Find median.

Partition elements into halves according to median.
Other sorting method.

Find median.

Partition elements into halves according to median.
Sort each half.
Other sorting method.

Find median.
- Partition elements into halves according to median.
- Sort each half.

\[ T(n) = 2T(n/2) + O(n). \]
Other sorting method.

Find median.

- Partition elements into halves according to median.
- Sort each half.

\[ T(n) = 2T\left(\frac{n}{2}\right) + O(n). \]
\[ T(n) = O(n \log n). \]
Randomized sorting method: quicksort.

Choose a random element to be partition elt, \( p \).
Split into larger and smaller elements based on \( p \).
Randomized sorting method: quicksort.

Choose a random element to be partition elt, \( p \).
  Split into larger and smaller elements based on \( p \).
Recurse on each group.
Randomized sorting method: quicksort.

Choose a random element to be partition elt, \( p \).
Split into larger and smaller elements based on \( p \).
Recurse on each group.

\[ T(n) = \text{???} \]
Randomized sorting method: quicksort.

Choose a random element to be partition elt, $p$.
   Split into larger and smaller elements based on $p$.

Recurse on each group.

$$T(n) = ??$$

How do you analyse this?
Randomized sorting method: quicksort.

Choose a random element to be partition elt, \( p \).
Split into larger and smaller elements based on \( p \).
Recurse on each group.

\[ T(n) = ??? \]

How do you analyse this?
Master Theorem doesn’t apply...
Randomized sorting method: quicksort.

Choose a random element to be partition elt, \( p \).
Split into larger and smaller elements based on \( p \).

Recurse on each group.

\[ T(n) = \text{???} \]

How do you analyse this?
Master Theorem doesn’t apply...
Also, how does one do average time?
Randomized sorting method: quicksort.

Choose a random element to be partition elt, \( p \).
Split into larger and smaller elements based on \( p \).
Recurse on each group.

\[ T(n) = \ldots \]

How do you analyse this?
Master Theorem doesn’t apply...
Also, how does one do average time?
Expected Time of algorithm on \( n \) elements.
One method: amortized analysis.

From the point of view of an element.
One method: amortized analysis.

From the point of view of an element.
Begin: size of elt’s group is $n$. 

$X$ - number of iterations for group to be small enough.

We know $X \sim G(\frac{1}{2})$.

We did this before.

$E[X] = \frac{1}{2} \times (1 + E[X]) + \frac{1}{2} \times (1)$

or $E[X] = 2$.

How much expected work on this element?

$E[n] = 2 + E[\frac{3}{4}n]$ 

Unfold:

$E[n] = 2 + 2 + \cdots + 2$

The number of 2's is $\log_\frac{4}{3} n$.

So, $O(\log n)$ per element, or $O(n \log n)$ overall elements.
One method: amortized analysis.

From the point of view of an element.
Begin: size of elt’s group is \( n \).
Each iteration: size of elt’s group gets smaller.
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is \( n \).

Each iteration: size of elt’s group gets smaller.

\[ \text{Size of the group become } \leq 3n/4 \text{ if ...} \]
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is $n$.

Each iteration: size of elt’s group gets smaller.

Size of the group become $\leq \frac{3n}{4}$ if ...

an elt in middle fourth is partition elt.
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is \( n \).

Each iteration: size of elt’s group gets smaller.

Size of the group become \( \leq 3n/4 \) if ...

an elt in middle fourth is partition elt.

\[ \begin{align*}
X & \quad \text{number of iterations for group to be small enough.} \\
\text{We know } & \quad X \sim G(1/2) \text{.} \\
E[X] & = \frac{1}{2} \times (1 + E[X]) + \frac{1}{2} \times (1) \quad \text{or} \quad E[X] = 2. \\
E[\frac{n}{E[X]}] & = 2 + E[3/4 n] \\
\text{Unfold: } & \quad E[\frac{n}{E[X]}] = 2 + 2 + \cdots + 2 \quad \text{\# of 2's is } \log_4 3n/4. \\
\text{So, } & \quad O(\log n) \text{ per element,} \\
\text{or } & \quad O(n \log n) \text{ overall elements.}
\end{align*} \]
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is $n$.

Each iteration: size of elt’s group gets smaller.

Size of the group become $\leq 3n/4$ if ...

an elt in middle fourth is partition elt.

$X$ - number of iterations for group to be small enough.
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is \( n \).

Each iteration: size of elt’s group gets smaller.

Size of the group become \( \leq 3n/4 \) if ...

an elt in middle fourth is partition elt.

\( X \) - number of iterations for group to be small enough.
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is $n$.

Each iteration: size of elt’s group gets smaller.

Size of the group become $\leq 3n/4$ if ...

an elt in middle fourth is partition elt.

$X$ - number of iterations for group to be small enough.

We know $X \sim G(\frac{1}{2})$. 
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is $n$.

Each iteration: size of elt’s group gets smaller.

Size of the group become $\leq 3n/4$ if ...

an elt in middle fourth is partition elt.

$X$ - number of iterations for group to be small enough.

We know $X \sim G(\frac{1}{2})$.

We did this before.
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is $n$.

Each iteration: size of elt’s group gets smaller.

Size of the group become $\leq 3n/4$ if ...

an elt in middle fourth is partition elt.

$X$ - number of iterations for group to be small enough.

We know $X \sim G(\frac{1}{2})$.

We did this before.

$E[X] = \frac{1}{2} \times (1 + E(X)) + \frac{1}{2} \times (1)$
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is $n$.

Each iteration: size of elt’s group gets smaller.

Size of the group become $\leq 3n/4$ if ...

an elt in middle fourth is partition elt.

$X$ - number of iterations for group to be small enough.

We know $X \sim G(\frac{1}{2})$.

We did this before.

$E[X] = 1/2 \times (1 + E(X)) + 1/2 \times (1)$

or $E[X] = 2$. 

How much expected work on this element?

$E[n] = 2 + E[3/4n]$ 

Unfold:

$E[n] = 2 + 2 + \cdots + 2$

The number of 2’s is $\log_4 3n$.

So, $O(\log n)$ per element, or $O(n \log n)$ overall elements.
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is $n$.

Each iteration: size of elt’s group gets smaller.

Size of the group become $\leq 3n/4$ if ...

... an elt in middle fourth is partition elt.

$X$ - number of iterations for group to be small enough.

We know $X \sim G(\frac{1}{2})$.

We did this before.

$E[X] = \frac{1}{2} \times (1 + E(X)) + \frac{1}{2} \times (1)

or $E[X] = 2$.

How much expected work on this element?
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is $n$.

Each iteration: size of elt’s group gets smaller.

    Size of the group become $\leq 3n/4$ if ...
    an elt in middle fourth is partition elt.

$X$ - number of iterations for group to be small enough.

We know $X \sim G\left(\frac{1}{2}\right)$.

We did this before.

$$E[X] = \frac{1}{2} \times (1 + E(X)) + \frac{1}{2} \times (1)$$

or $E[X] = 2$.

How much expected work on this element?

$$E[n] = 2 + E[3/4n]$$
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is \( n \).

Each iteration: size of elt’s group gets smaller.

Size of the group become \( \leq \frac{3n}{4} \) if ...

an elt in middle fourth is partition elt.

\( X \) - number of iterations for group to be small enough.

We know \( X \sim G\left(\frac{1}{2}\right) \).

We did this before.

\[ E[X] = \frac{1}{2} \times (1 + E(X)) + \frac{1}{2} \times (1) \]

or \( E[X] = 2 \).

How much expected work on this element?

\[ E[n] = 2 + E[\frac{3}{4}n] \]

Unfold: \( E[n] = 2 + 2 + \cdots + 2 \)
One method: amortized analysis.

From the point of view of an element.
Begin: size of elt’s group is \( n \).
Each iteration: size of elt’s group gets smaller.

Size of the group become \( \leq 3n/4 \) if ...
    an elt in middle fourth is partition elt.

\( X \) - number of iterations for group to be small enough.
We know \( X \sim G(\frac{1}{2}) \).
We did this before.
\[
E[X] = \frac{1}{2} \times (1 + E(X)) + \frac{1}{2} \times (1)
\]
or \( E[X] = 2 \).

How much expected work on this element?
\[
E[n] = 2 + E[3/4n]
\]
Unfold: \( E[n] = 2 + 2 + \cdots + 2 \)
The number of 2’s is \( \log_{4/3} n \).
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is $n$.

Each iteration: size of elt’s group gets smaller.

Size of the group become $\leq \frac{3n}{4}$ if ... an elt in middle fourth is partition elt.

$X$ - number of iterations for group to be small enough.
We know $X \sim G(\frac{1}{2})$.
We did this before.

$E[X] = \frac{1}{2} \times (1 + E(X)) + \frac{1}{2} \times (1)$

or $E[X] = 2$.

How much expected work on this element?

$E[n] = 2 + E[\frac{3}{4}n]$  
Unfold: $E[n] = 2 + 2 + \cdots + 2$

The number of 2’s is $\log_{4/3} n$.
So, $O(\log n)$ per element,
One method: amortized analysis.

From the point of view of an element.

Begin: size of elt’s group is $n$.

Each iteration: size of elt’s group gets smaller.

Size of the group become $\leq 3n/4$ if ...

an elt in middle fourth is partition elt.

$X$ - number of iterations for group to be small enough.
We know $X \sim G(\frac{1}{2})$.
We did this before.

$$E[X] = \frac{1}{2} \times (1 + E(X)) + \frac{1}{2} \times (1)$$

or $E[X] = 2$.

How much expected work on this element?

$$E[n] = 2 + E[3/4n]$$

Unfold: $E[n] = 2 + 2 + \cdots + 2$

The number of 2’s is $\log_{4/3} n$.
So, $O(\log n)$ per element, or $O(n \log n)$ overall elements.
Conclusion: More divide and conquer.

You came up with iterative,
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$.
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$. 
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$.

Mergesort: $O(n \log n)$. 
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$.

Mergesort: $O(n\log n)$.
  Divide in two, sort each, merge.

Median: $O(n)$ expected time.
  Do selection.
  Pick random partition, recurse on the correct side.
  Analysis: after an average of two steps, problem smaller by $3/4$.
  Geometric series of runtimes.

Quicksort: $O(n\log n)$ expected time.
  Random partition.
  Work per element is constant in each iteration.
  Expected $O(\log n)$ iterations.
  $O(n\log n)$ work for $n$ elements.
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$.

Mergesort: $O(n \log n)$.
  Divide in two, sort each, merge.
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: \(O(n^2)\).

Mergesort: \(O(n \log n)\).
   Divide in two, sort each, merge.

Median: \(O(n)\) expected time.
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$.

Mergesort: $O(n\log n)$.
  Divide in two, sort each, merge.

Median: $O(n)$ expected time.
  Do selection.
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$.

Mergesort: $O(n \log n)$.
  Divide in two, sort each, merge.

Median: $O(n)$ expected time.
  Do selection.
  Pick random partition, recurse on the correct side.
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$.

Mergesort: $O(n \log n)$.
   Divide in two, sort each, merge.

Median: $O(n)$ expected time.
   Do selection.
   Pick random partition, recurse on the correct side.
   Analysis: after an average of two steps, problem smaller by $3/4$. 
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$.

Mergesort: $O(n \log n)$.
  Divide in two, sort each, merge.

Median: $O(n)$ expected time.
  Do selection.
  Pick random partition, recurse on the correct side.
  Analysis: after an average of two steps, problem smaller by $3/4$.
  Geometric series of runtimes.
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$.

Mergesort: $O(n \log n)$.
   Divide in two, sort each, merge.

Median: $O(n)$ expected time.
   Do selection.
   Pick random partition, recurse on the correct side.
   Analysis: after an average of two steps, problem smaller by $3/4$.
   Geometric series of runtimes.

Quicksort: $O(n \log n)$ expected time.
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$.

Mergesort: $O(n \log n)$.
   Divide in two, sort each, merge.

Median: $O(n)$ expected time.
   Do selection.
   Pick random partition, recurse on the correct side.
   Analysis: after an average of two steps, problem smaller by \( \frac{3}{4} \).
   Geometric series of runtimes.

Quicksort: $O(n \log n)$ expected time.
   Random partition.
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$.

Mergesort: $O(n \log n)$.
   Divide in two, sort each, merge.

Median: $O(n)$ expected time.
   Do selection.
   Pick random partition, recurse on the correct side.
   Analysis: after an average of two steps, problem smaller by $3/4$.
   Geometric series of runtimes.

Quicksort: $O(n \log n)$ expected time.
   Random partition.
   Work per/element is constant in each iteration.
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: $O(n^2)$.

Mergesort: $O(n \log n)$.
   Divide in two, sort each, merge.

Median: $O(n)$ expected time.
   Do selection.
   Pick random partition, recurse on the correct side.
   Analysis: after an average of two steps, problem smaller by $3/4$.
   Geometric series of runtimes.

Quicksort: $O(n \log n)$ expected time.
   Random partition.
   Work per/element is constant in each iteration.
   Expected $O(\log n)$ iterations.
Conclusion: More divide and conquer.

You came up with iterative, and divide and conquer algorithms.

Iterative: \( O(n^2) \).

Mergesort: \( O(n \log n) \).
  Divide in two, sort each, merge.

Median: \( O(n) \) expected time.
  Do selection.
    Pick random partition, recurse on the correct side.
    Analysis: after an average of two steps, problem smaller by \( \frac{3}{4} \).
      Geometric series of runtimes.

Quicksort: \( O(n \log n) \) expected time.
  Random partition.
  Work per/element is constant in each iteration.
  Expected \( O(\log n) \) iterations.
  \( O(n \log n) \) work for \( n \) elements.