Lecture 9: Time complexity analysis of sorting & k-selection

Xiaoqian Sun / Sebastian Wandelt

Beihang University

Outline

- K-selection problem
- Worst-case time complexity analysis of sorting
- Sorting in linear-time

k-Selection

Recap

Select-k Algorithm

In the `select_k` algorithm, we will attempt to return the k^{th} smallest element of an unsorted list of values **A**.

41	**23**	**11**	**5**	**22**	**4**	**3**	**14**	**52**	**20**

```
select_k(A,0) => 3      select_k(A,0) => min(A)
select_k(A,4) => 14     select_k(A,⌈n/2⌉-1) => median(A)
select_k(A,9) => 52     select_k(A,n-1) => max(A)
```

A Slower Select-k Algorithm

```
algorithm naive_select_k(list A, k):
  A = mergesort(A)
  return A[k]
```

Runtime: ???

A Slower Select-k Algorithm

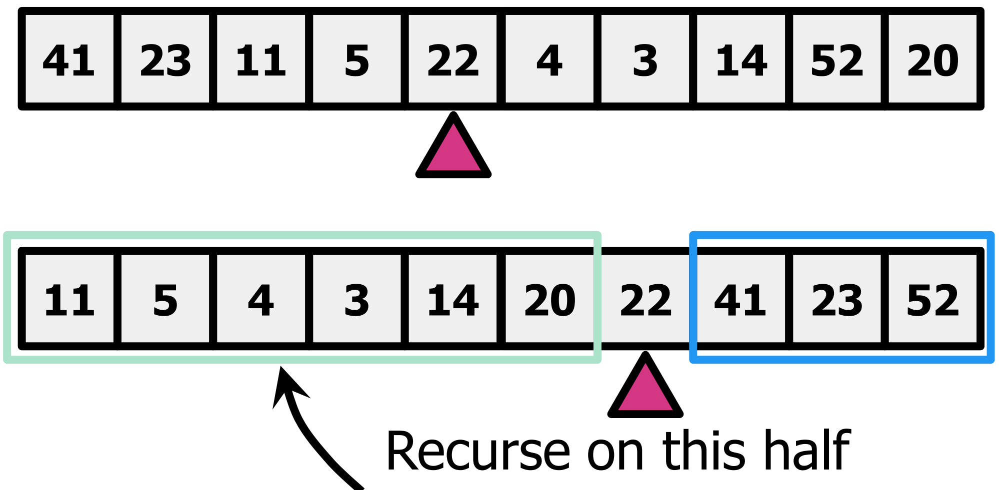
```
algorithm naive_select_k(list A, k):
  A = mergesort(A)
  return A[k]
```

Runtime: `O(nlogn)`

A “better” Select-k Algorithm

Main idea: choose a pivot, partition around it, and recurse.

Suppose we call `select_k(A,3)`.



Randomly (for now) choose 22 to be the pivot.

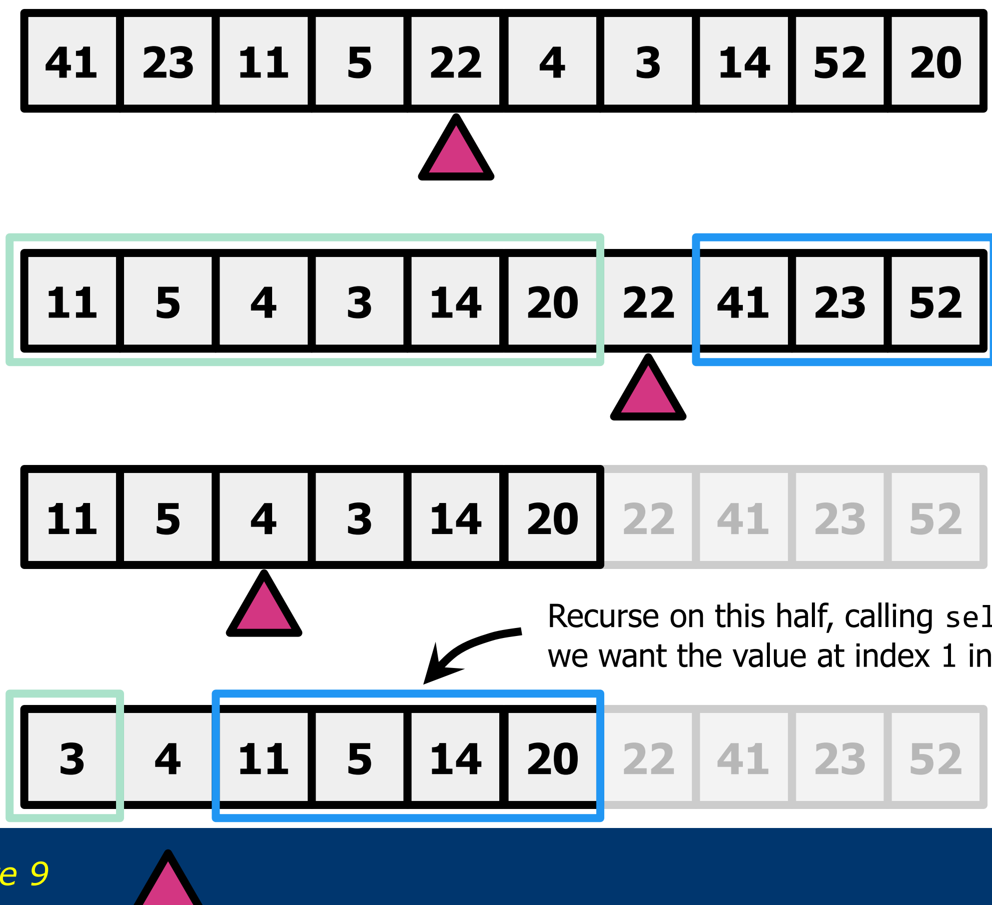
Partition around 22, such that all values to its left are less than it and all values to its right are greater than it.

Recurse on this half since 22 occupies index 6 and 3 < 6, calling select_k(A,3)

Select-k Algorithm

Main idea: choose a pivot, partition around it, and recurse.

Suppose we call `select_k(A,3)`.



Randomly (for now) choose 22 to be the pivot.

Partition around 22, such that all values to its left are less than it and all values to its right are greater than it.

Randomly (for now) choose 4 to be the pivot.

Recurse on this half, calling `select_k(A[2:],1)` since we want the value at index 1 in the right list.

Partition around 4.

Partitioning

```
algorithm partition(list A, p):
  L, R = []
  for i = 0 to length(A)-1:
    if i == p: continue
    else if A[i] <= A[p]:
      L.append(A[i])
    else if A[i] > A[p]:
      R.append(A[i])
  return L, A[p], R
```

Runtime: $O(n)$

Select-k Algorithm

```
algorithm select_k(list A, k):
  if length(A) == 1: return A[0]
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

Overall Runtime: ???

Select-k Algorithm

```
algorithm select_k(list A, k):
  if length(A) == 1: return A[0]
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

Runtime: $O(n^2)$ ← Why was that?

Summary

- The naïve algorithm needed O(n*log n)
  - Mergesort and retrieve k-th smallest by lookup
- The "improved" algorithm needed O(n*n)
  - Divide and conquer with pivot-selection
- What did we gain?

k-Selection

in O(n)

A new pivot selection strategy!

- Recall the structure of our select-k algorithm:

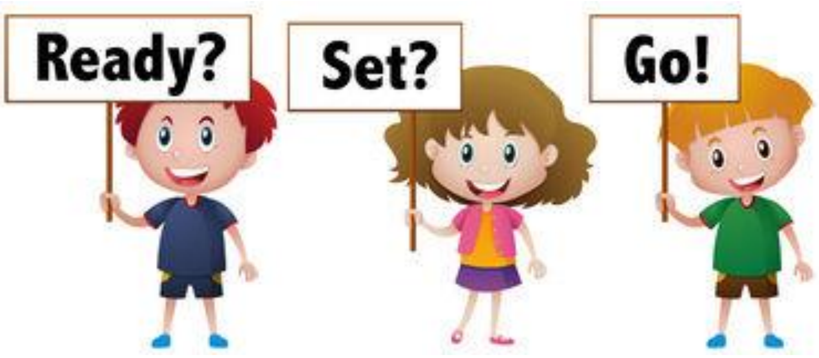
  select_k => choose_pivot

          select_k => choose_pivot

                  select_k => choose_pivot

                          ….

- Any ideas?



A new pivot selection strategy!

- The key is to find a good pivot element, to partition the list into two **equally sized** sublists!
- Divide and conquer with $T(n)=T(n/2)+X$
  – Likely to be linear with the length of the list n, depending on X

Select-k Algorithm

```
algorithm smartly_choose_pivot(list A):
  groups = split A into m=⌈length(A)/5⌉
           groups, of size ≤ 5 each
  candidate_pivots = []
  for i = 0 to m-1:
    p_i = median(groups[i])  # O(1)
    candidate_pivots.append(p_i)
  A[p] = select_k(candidate_pivots, m/2)
  return index_of(A[p])
```

Select-k Algorithm


```
algorithm smartly_choose_pivot(list A):
  groups = split A into m=⌈length(A)/5⌉
           groups, of size ≤ 5 each
  candidate_pivots = []
  for i = 0 to m-1:
    p_i = median(groups[i])  # O(1)
    candidate_pivots.append(p_i)
  A[p] = select_k(candidate_pivots, m/2)
  return index_of(A[p])
```

Why is that?

Select-k Algorithm

```
algorithm select_k(list A, k):
  if length(A) ≤ 100:
    return naive_select_k(A, k)
  p = smartly_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

Runtime: O(n)

But why? This is not obvious at all...

Analyzing Runtime

Instead of `p = random_choose_pivot(A)`, now we have `p = smartly_choose_pivot(A)`.

Why is this algorithm `O(n)`?

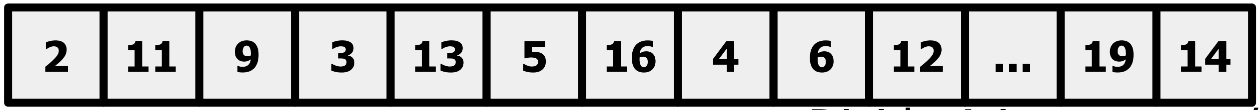
Analyzing Runtime

Instead of `p = random_choose_pivot(A)`, now we have `p = smartly_choose_pivot(A)`.

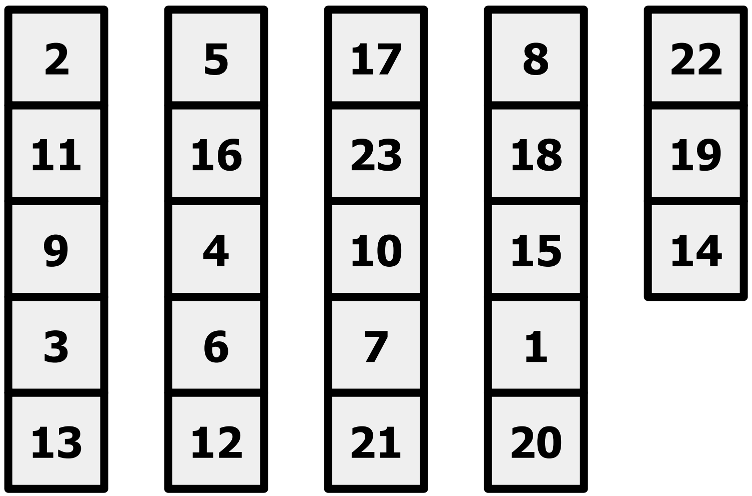
Why is this algorithm **`O(n)`**?

Main idea: each of the arrays L and R are pretty balanced. Thus, while the median of medians might not be the actual median, it's pretty close.

Analyzing Runtime



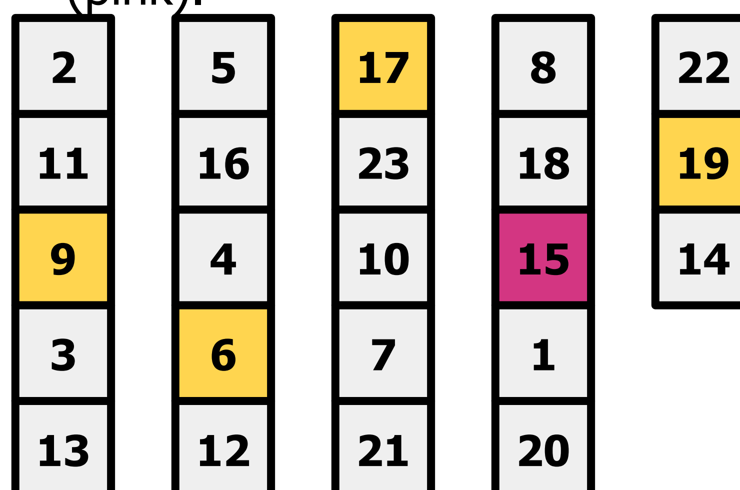
Divide A into $m = \lceil n/5 \rceil$ groups
of at most 5 elements



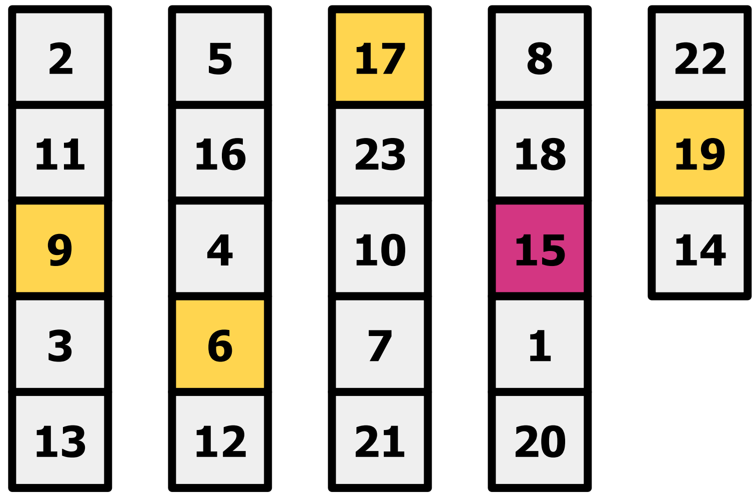
at most 5 elements

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

Find the median of each of the groups (yellow) and recursively call `select_k` to find the median of these medians (pink).



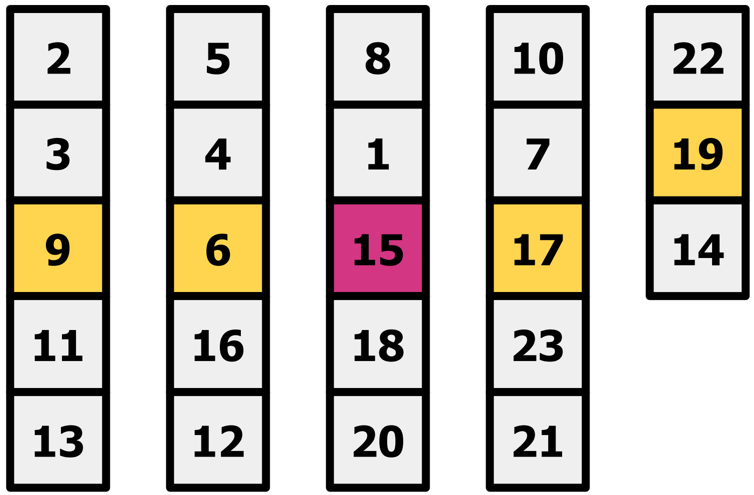
Analyzing Runtime



Clearly the median of medians (15) is not necessarily the actual median (12), but we claim that it's guaranteed to be pretty close.

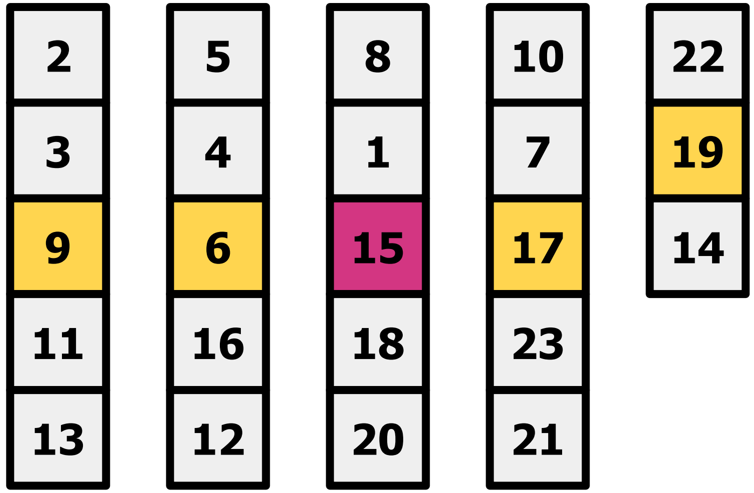
What is "pretty close". Quite fuzzy for now, we will check later!

Analyzing Runtime



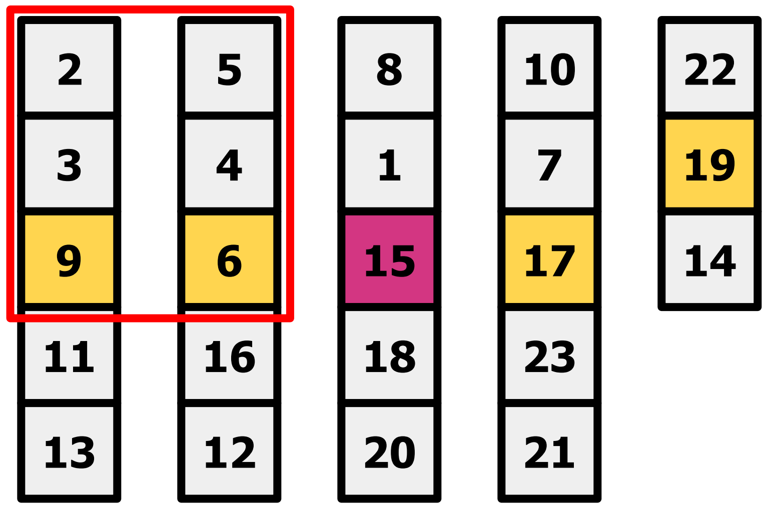
To see why, let’s partition elements within each of the groups around the group’s median, and partition the groups around the group with the median of medians.

Analyzing Runtime



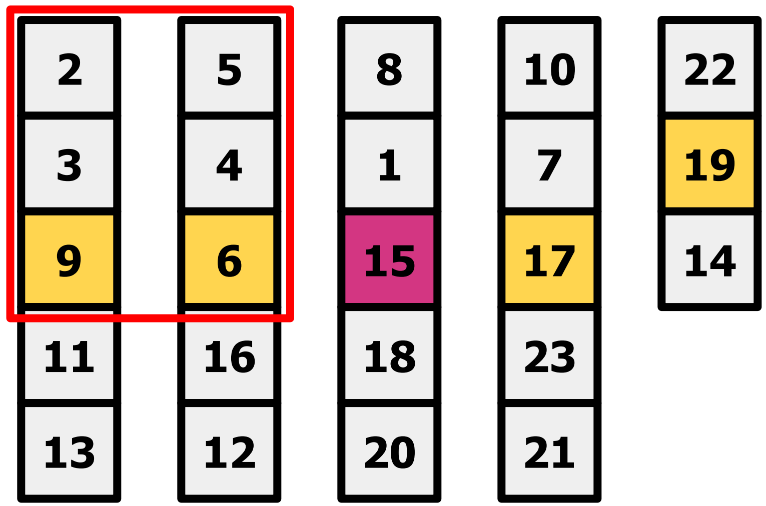
How many elements are smaller than the median of medians?

Analyzing Runtime



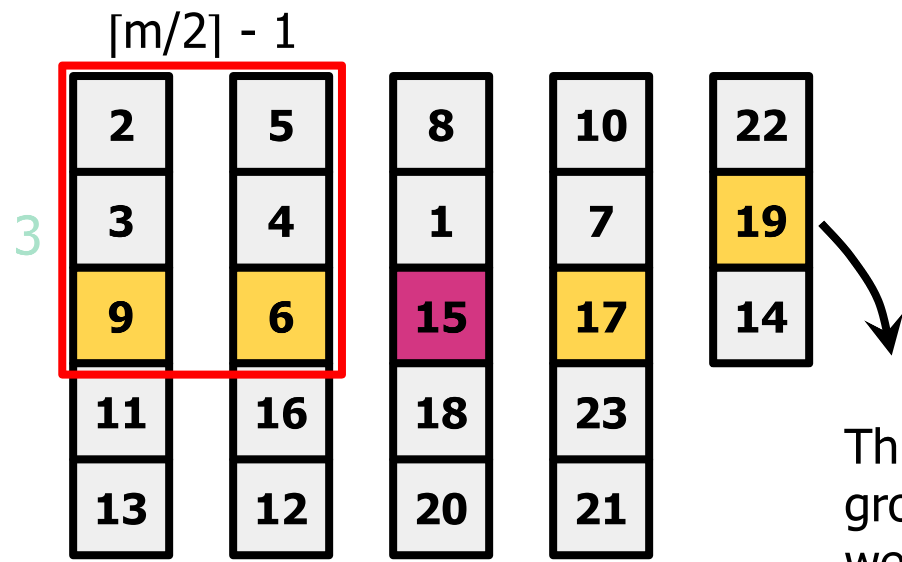
At least these guys (2, 3, 4, 5, 6, 9): everything above and to the left. There might be more (1, 7, 8, 11, 12, 13, 14), but we are guaranteed that *at least* these guys will be smaller.

Analyzing Runtime



How many are there?

Analyzing Runtime

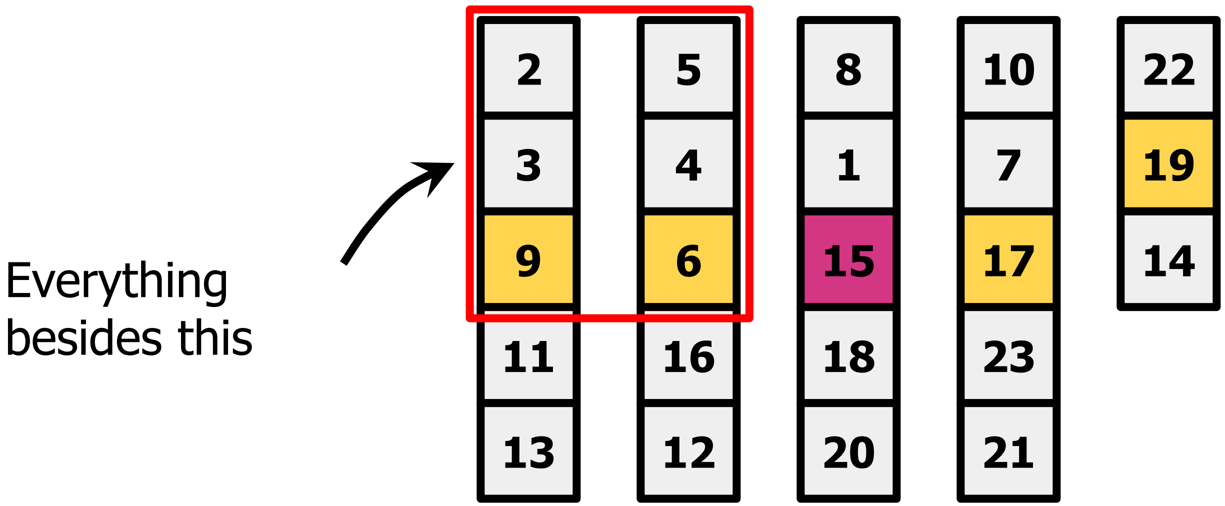


This is the "leftovers group"; if its contents were [12, 14, 16], then it would not have 3 elements less than the median of medians; thus the -1 below.

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

One of groups could have been the leftovers group

Analyzing Runtime



How many elements are larger than the median of medians?

At most $n - 1 - 3\cdot(\lceil m/2 \rceil - 1 - 1) \leq 7n/10 + 5$.

Analyzing Runtime

We just showed that …

$$3 \cdot (\lceil m/2 \rceil - 2) \leq |L|$$

`smartly_choose_pivot` will choose a pivot greater than at least $3 \cdot (\lceil m/2 \rceil - 2)$ elements.

$$|R| \leq 7n/10 + 5$$

`smartly_choose_pivot` will choose a pivot less than at most $7n/10 + 5$ elements.

Analyzing Runtime

We can just as easily show the inverse.

$$3 \cdot (\lceil m/2 \rceil - 2) \leq |L| \leq 7n/10 + 5$$

$$3 \cdot (\lceil m/2 \rceil - 2) \leq |R| \leq 7n/10 + 5$$

Analyzing Runtime

What's the greatest number of elements that can be smaller than p?

- `random_choose_pivot` might choose the largest element, so n-1.
- `smartly_choose_pivot` will choose an element greater than at most $7n/10 + 5$ elements.

What's the greatest number of elements that can be larger than p?

- `random_choose_pivot` might choose the smallest element, so n-1.
- `smartly_choose_pivot` will choose an element smaller than at most $7n/10 + 5$ elements.

Analyzing Runtime

Recurrence relation: $T(n) \leq c \cdot n + T(\lceil n/5 \rceil) + T(\lceil 7n/10 + 5 \rceil)$.

Partitioning, computing n/5 medians

Computing the median of n/5 medians.

Recursing on L or R.

But what if n = 4?

We introduce a "fat base case" where $T(n) = \Theta(1) \leq c$ for $n \leq 100$.

Recall that the Master Method only works when the sub-problems are the same size.

To prove this recurrence relation yields a runtime of **O(n)**, we will employ substitution method.

Analyzing Runtime

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

Proof: We guess that for all $n \geq 1$, $T(n) \leq Cn$ for some C that we will determine later; this means $T(n) = O(n)$.

We proceed by induction. As a **base case**, if $1 \leq n \leq 100$, then $T(n) \leq c \leq Cn$ will be true as long as we pick $C \geq c$.

For the **inductive step**, assume for some $n \geq 100$ that the claim holds for all $1 \leq n' < n$. Note that $1 \leq \lceil n/5 \rceil, \lceil 7n/10 + 5 \rceil < n$. Then:

$$
\begin{aligned}
T(n) &\leq T(\lceil n/5 \rceil) + T(\lceil 7n/10 + 5 \rceil) + cn \\
&\leq C\lceil n/5 \rceil + C\lceil 7n/10 + 5 \rceil + cn \\
&= C(n/5 + 1) + C(7n/10 + 5 + 1) + cn \\
&= 9Cn/10 + 7C + cn \\
&= Cn + (7C + cn - Cn/10)
\end{aligned}
$$

If we pick $C = 50c$, then $7C + cn - Cn/10 \leq 0$ and $T(n) \leq Cn$ holds, completing the induction. ■

Substitution Method

To use substitution method, proceed as follows:

Make a guess of the form of your answer (e.g. Cn)

Proceed by induction to prove the bound holds, noting what constraints arise on your undetermined constants (e.g. C).

- If your induction succeeds, you will have values for your undetermined constants.

- If the induction fails, then it doesn't necessarily imply that your guess fails to bound the recurrence.

Open question

- Why did we choose the median over groups of size 5?
  - Could we have chosen any other number?

Open question

- Why did we choose the median over groups of size 5?
  - Could we have chosen any other number?
- We could:
  - An odd number, since it is simpler to denote median
  - A number larger than three, otherwise the boundary estimation does not hold, and the runtime is in O(n*log n)

Open question

- Could we have partitioned the list into 5 sublists of length n/5?

Open question

- Could we have partitioned the list into 5 sublists of length n/5?
- No, because then, finding the median is not a constant time operation!

Sorting revisited

Comparison-Based Sorting

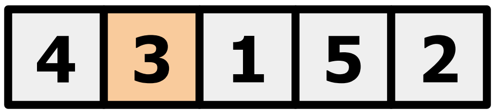
These algorithms use "comparisons" to achieve their output.

`insertion_sort` and `mergesort` are comparison-based sorting algorithms.

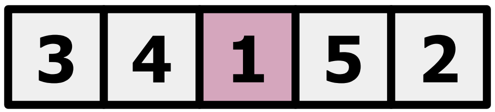
`select_k` is a comparison-based algorithm.

A *comparison* compares two values. e.g. Is **A[0]** < **A[1]**? Is **A[0]** < **A[4]**?

Recall, insertion sort.



Is 3 < 4?



Is 1 < 4? Is 1 < 3?

Comparison-Based Sorting

Theorem: Any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$-time.

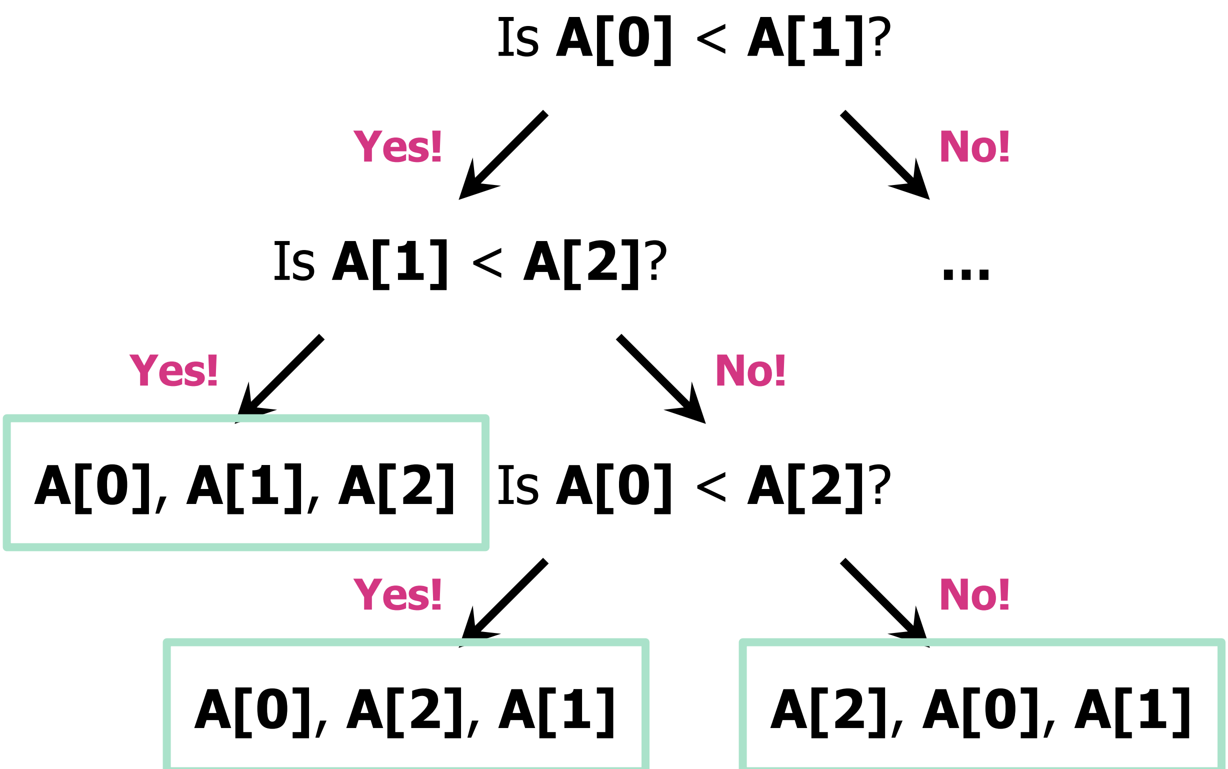
Proof:

Hmm …

Comparison-Based Sorting

We can represent the comparisons made by a comparison-based sorting algorithm as a decision tree.

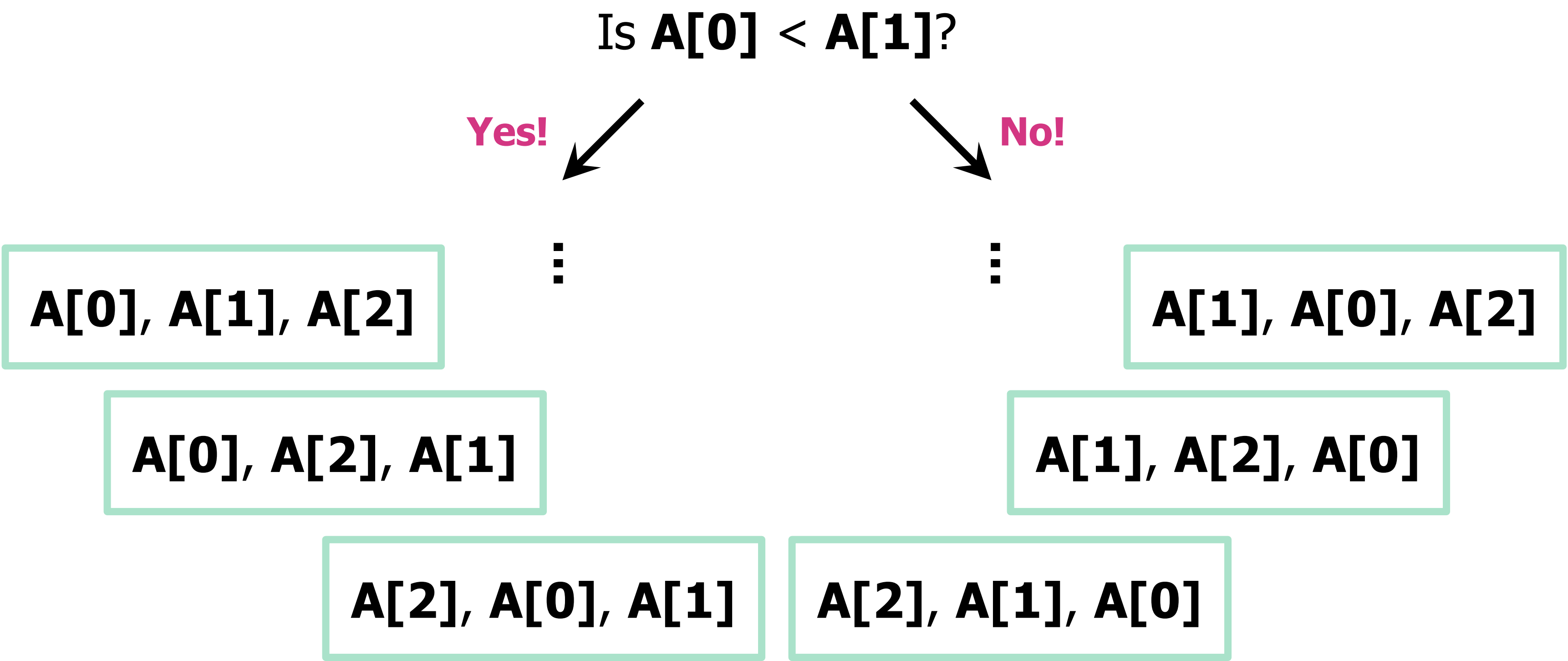
Suppose we want to sort three items in a list **A** with three elements.



Comparison-Based Sorting

The leaves are all of the possible orderings of the items.

The worst-case runtime must be at least
Ω(length of the longest path).



Comparison-Based Sorting

How long is the longest path?

- At least how many leaves must this decision tree have?
- What is the depth of the shallowest tree with this many leaves?

Comparison-Based Sorting

How long is the longest path?

- At least how many leaves must this decision tree have? **n!**
- What is the depth of the shallowest tree with this many leaves? **log(n!)**

The longest path is at least log(n!), so the worst-case runtime must be at least **Ω(log(n!))** = **Ω(n log(n))**.

Comparison-Based Sorting

Theorem: Any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$-time.

Proof:

Any deterministic comparison-based sorting algorithm can be represented as a decision tree with $n!$ leaves.

The worst-case runtime is at least the depth of the decision tree.

All decision trees with $n!$ leaves have depth $\Omega(n \log(n))$.

Therefore, any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$-time

Beyond Comparisons

But then what's this nonsense about linear-time sorting algorithms?

We achieve O(n) worst-runtime if we make **assumptions on the input**, e.g., the input consists of integers that range from 0 to k-1 only.

Counting sort

```
algorithm counting_sort(A, k):
  # A consists of n ints, ranging from
  # 0 to k-1
  counts = [0 * k]  # list of k zeros
  for a_i in A:
    counts[a_i] += 1
  result = []
  for a_i = 0 to length(counts)-1:
    append counts[a_i] a_i's to results
  return results
```

Runtime: O(n+k)

Counting sort

Suppose **A** consists of 8 integers ranging from 0 to 3.

```
counting_sort(A, 4)
```

0	0	3	1	1	3	1	0

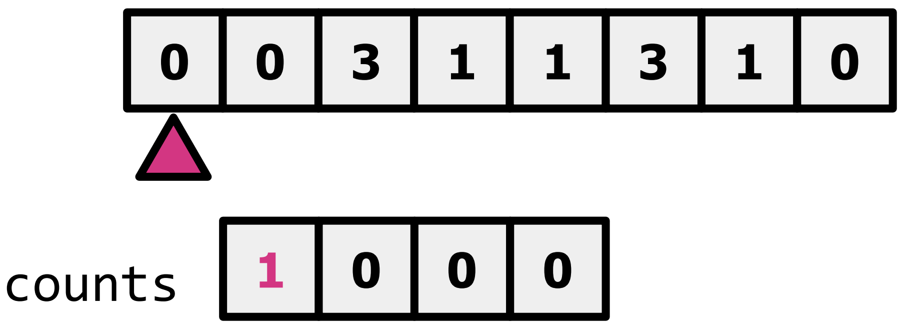
counts

0	0	0	0

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

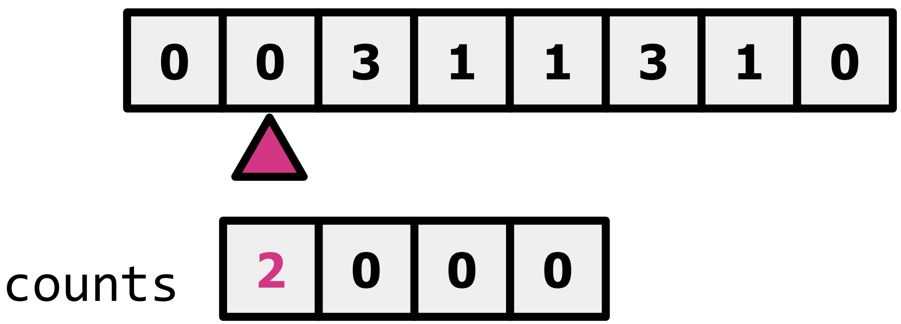
```
counting_sort(A, 4)
```



Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

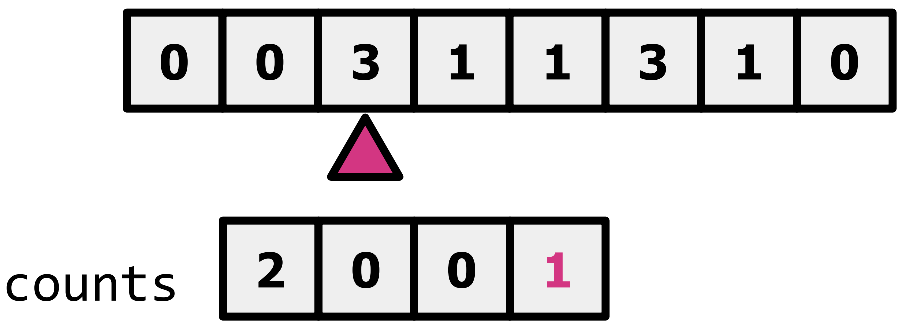
```
counting_sort(A, 4)
```



Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

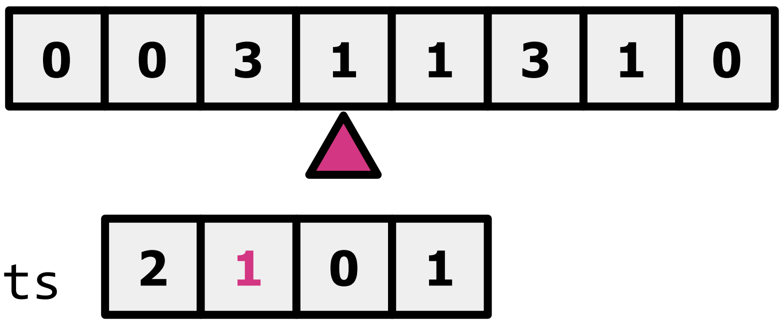
```
counting_sort(A, 4)
```



Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



counts

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

`counting_sort(A, 4)`

0	0	3	1	1	3	1	0

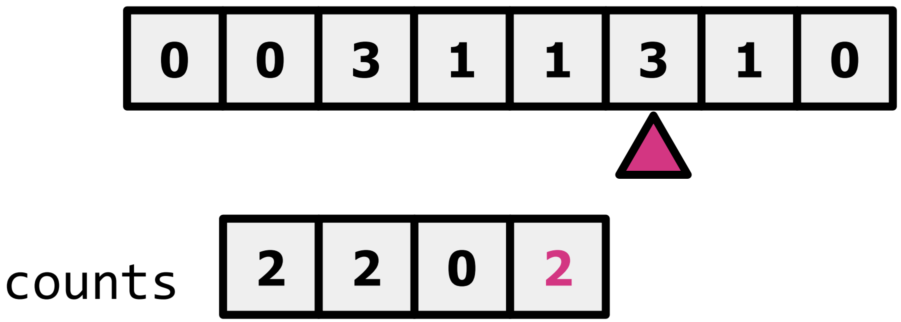
counts

2	2	0	1

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

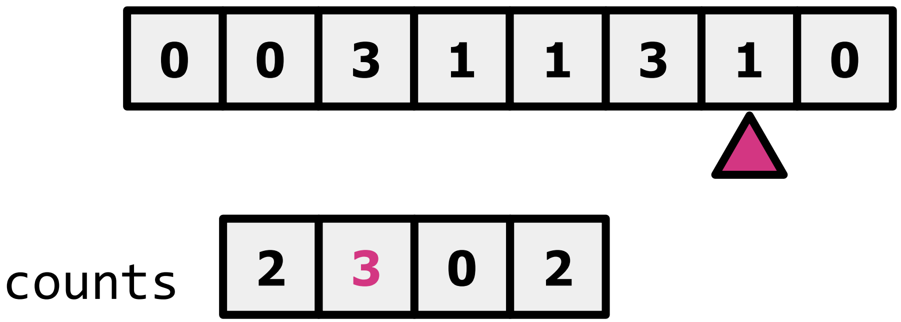
```
counting_sort(A, 4)
```



Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```

0	0	3	1	1	3	1	0

counts

3	3	0	2

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```

0	0	3	1	1	3	1	0

`counts`

3	3	0	2

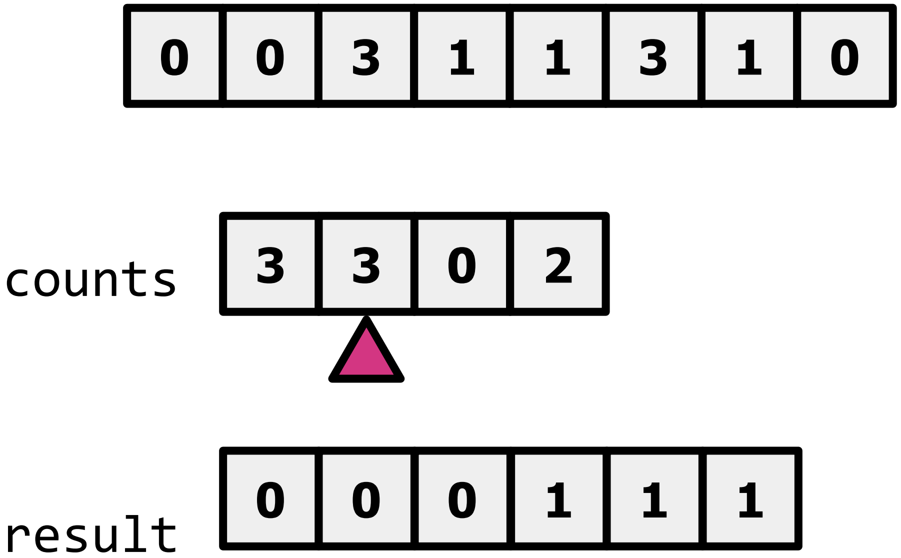
`result`

0	0	0

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```

0	**0**	**3**	**1**	**1**	**3**	**1**	**0**

`counts`

3	**3**	**0**	**2**

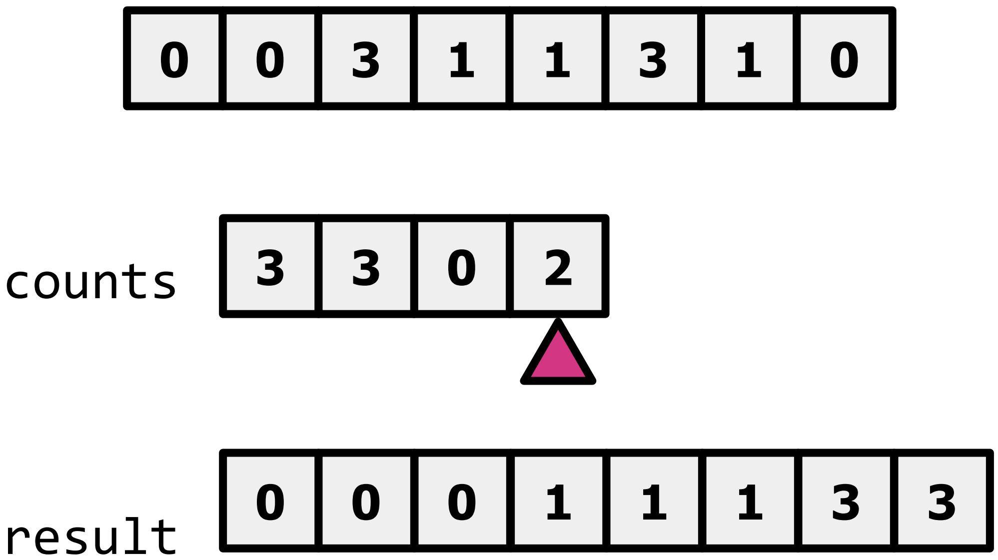
`result`

0	**0**	**0**	**1**	**1**	**1**

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



Main insight

- All operations can be performed in linear time
- Now we will look at a generalization of Counting Sort:
  - Bucket Sort
    - Assume that we have a limited number of buckets
    - Each bucket can be sorted independently

Bucket sort

```
algorithm bucket_sort(A, k, num_buckets):
  # A consists of n numbers ranging from 0 to k-1
  buckets = [[] * num_buckets]
  for x in A:
    buckets[x*num_buckets/k].append(x)
  if num_buckets < k:
    for bucket in buckets:
       sort(bucket) by their keys
  result = concatenate buckets by their values
  return result
```

Runtime: ???

Bucket sort

```
algorithm bucket_sort(A, k, num_buckets):
  # A consists of n numbers ranging from 0 to k-1
  buckets = [[] * num_buckets]
  for x in A:
    buckets[x*num_buckets/k].append(x)
  if num_buckets < k:
    for bucket in buckets:
       sort(bucket) by their keys
  result = concatenate buckets by their values
  return result
```

Runtime: $O(n+k)$ or $O(n\log n)$

Only guaranteed if num_buckets >= k

Bucket sort

Two cases for k and num_buckets in `bucket_sort`:

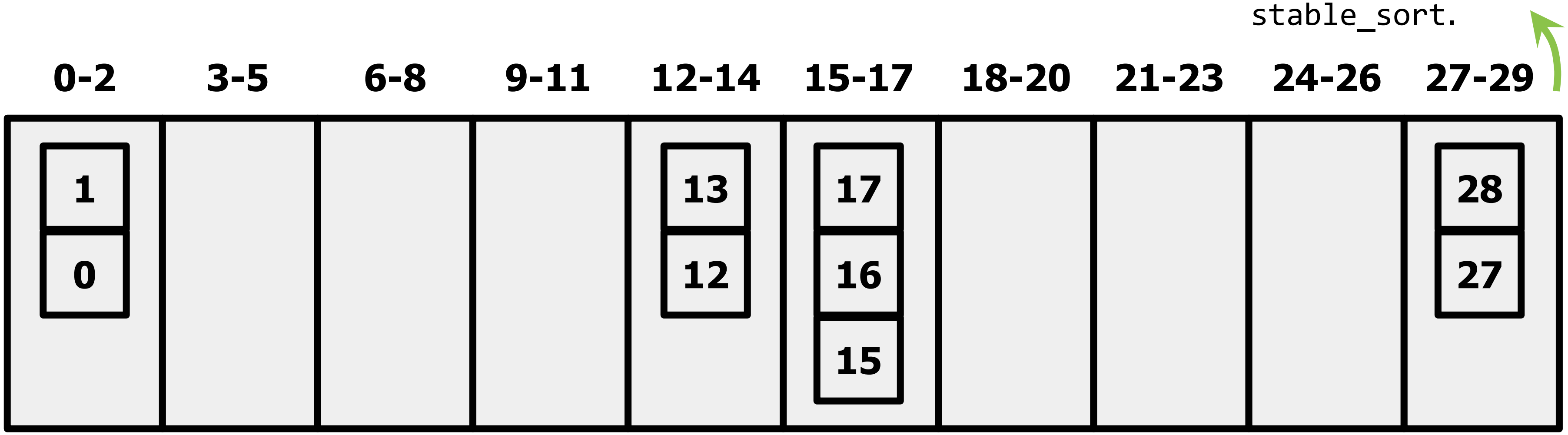
(1) k ≤ num_buckets: At most one key per bucket, so buckets do not require an additional `stable_sort` to be sorted (similar to `counting_sort`).

(2) k > num_buckets: Maybe multiple keys per bucket, so buckets require an additional `stable_sort` to be sorted.

Suppose k = 30 and num_buckets = 10. Then we group keys 0 to 2 in the same bucket, 3 to 5 in the same bucket, etc.

A= [17, 13, 16, 12, 15, 1, 28, 0, 27] produces:

Only the keys in the (key, value) pairs are shown here, and all of the buckets require `stable_sort`.



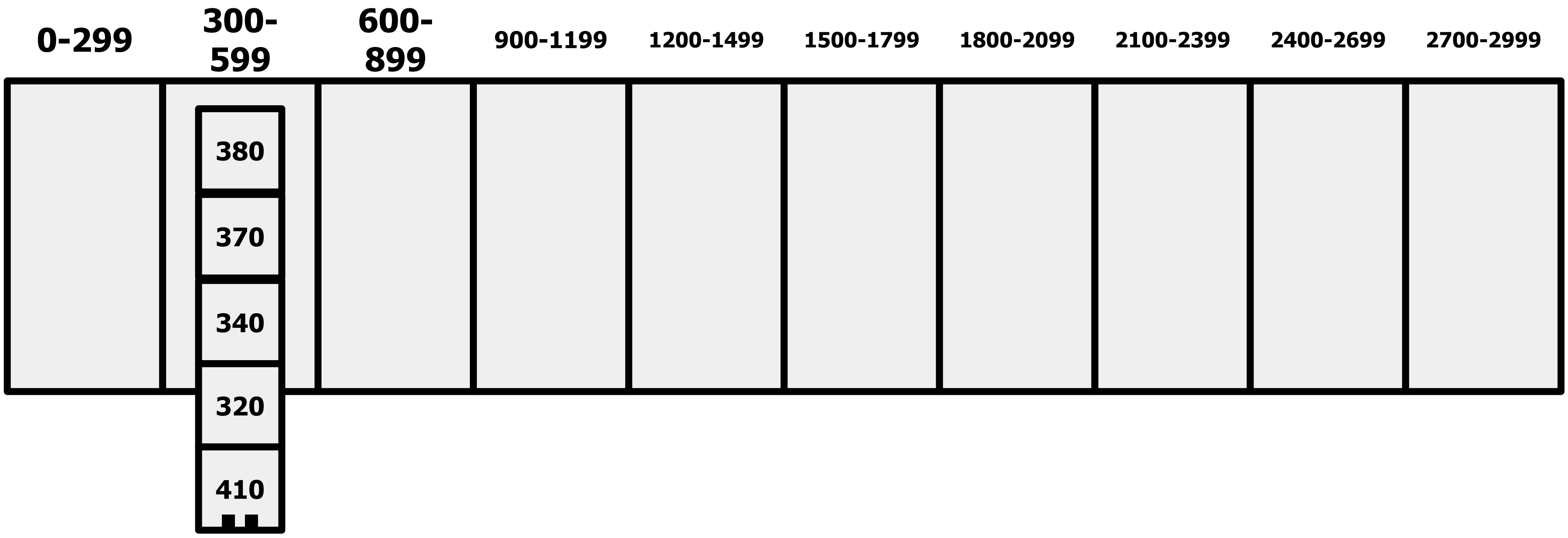
Bucket sort, case (2)

Why `O(nlogn)` in case (2)?

With multiple keys per bucket, a bucket might receive all of the inserted keys.

Suppose the `bucket_sort` caller specifies k = 3000 and num_buckets = 10, but then inserts elements all from the same bucket.

A = [380, 370, 340, 320, 410, …] would need to `stable_sort` all of the elements in the original list since they all fall in the same bucket.



Thank you very much!