Introduction to Algorithms

L4. Linear time sorting

Instructor : Kilho Lee

Today's Outline

- Quick sort
- Comparison-based sorting lower bounds
- Linear-Time Sorting
    - *Algorithms: Counting sort, bucket sort, and radix sort*
    - Reading: CLRS 8.1-8.2

(Randomized) Quicksort

Quicksort

It behaves as follows:

- If the list has 0 or 1 elements it's sorted.
- Otherwise, choose a pivot and partition around it.
- Recursively apply `quicksort` to the sublists to the left and right of the pivot.

Quicksort

0	11	7	4	8	3	2	9	6	10	5	1

Choose a pivot.

Quicksort

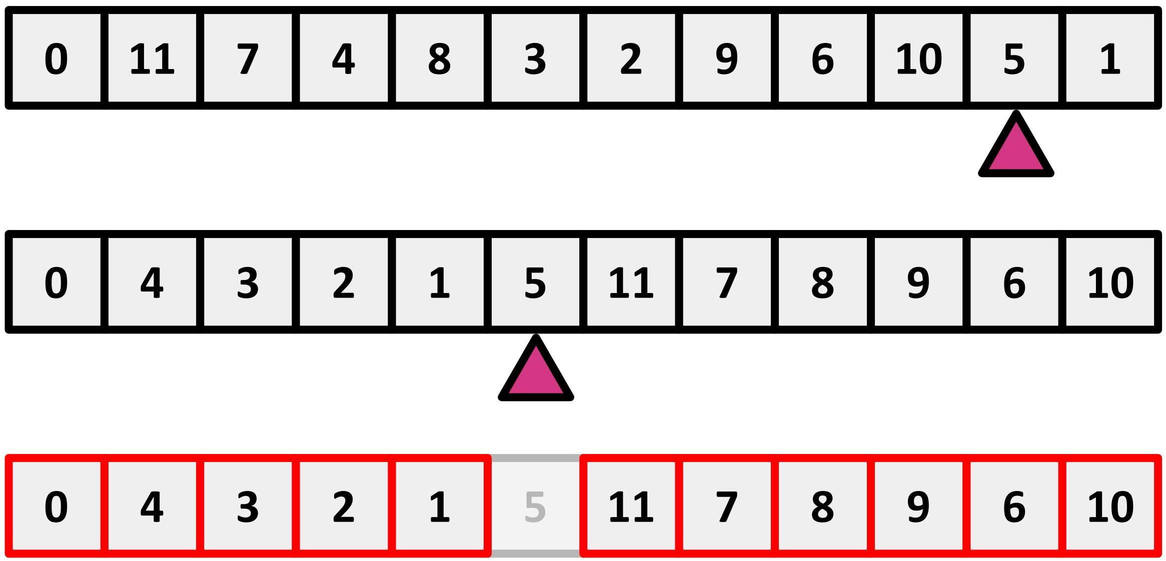
0	11	7	4	8	3	2	9	6	10	5	1

Choose a pivot.

0	4	3	2	1	5	11	7	8	9	6	10

Partition around it.

Quicksort



Choose a pivot.

Partition around it.

Recurse on both subarrays.

Quicksort

0	11	7	4	8	3	2	9	6	10	5	1

Choose a pivot.

0	4	3	2	1	5	11	7	8	9	6	10

Partition around it.

0	4	3	2	1	5	11	7	8	9	6	10

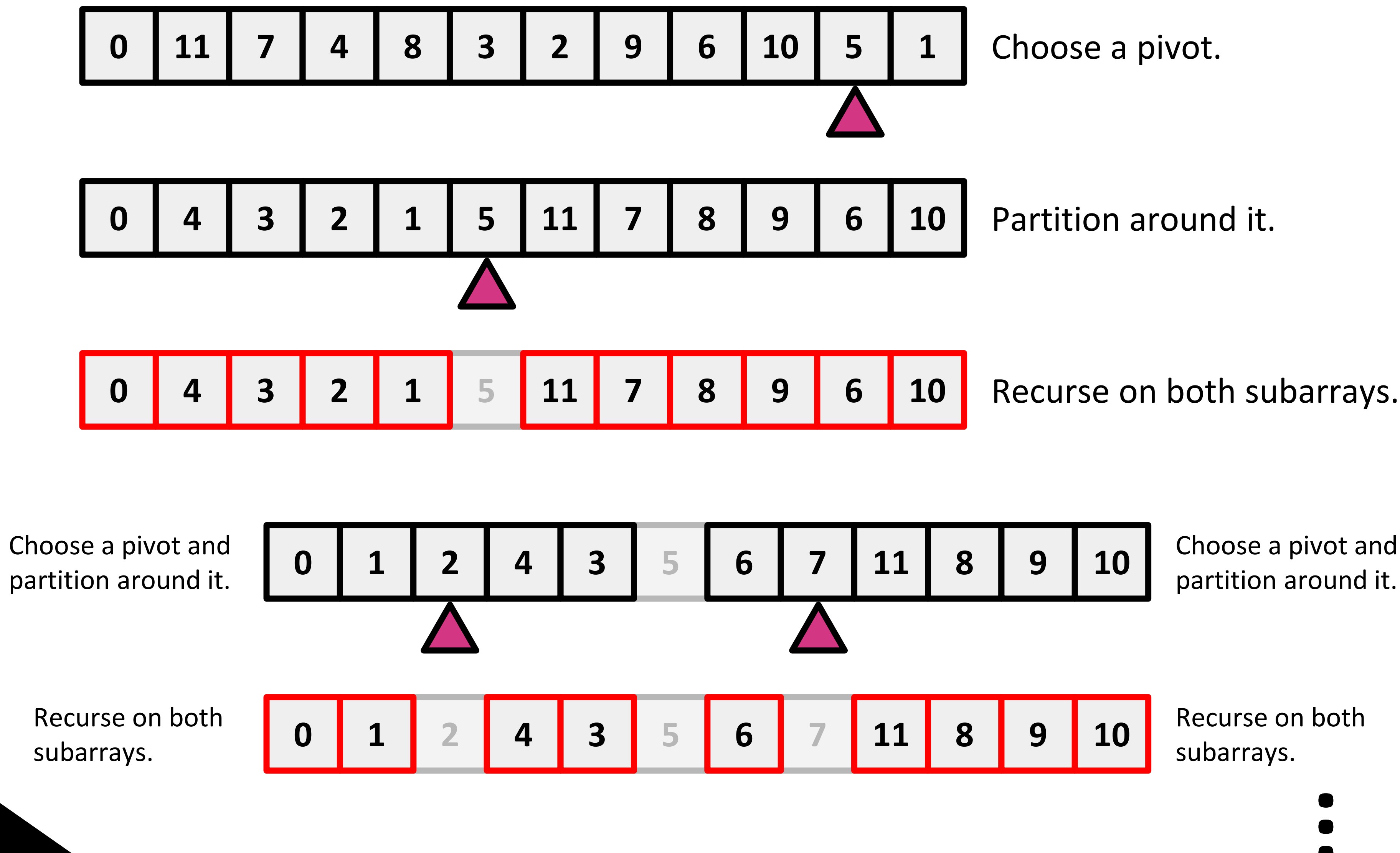
Recurse on both subarrays.

Choose a pivot and partition around it.

0	1	2	4	3	5	6	7	11	8	9	10

Choose a pivot and partition around it.

Quicksort


0 11 7 4 8 3 2 9 6 10 5 1
Choose a pivot.
0 4 3 2 1 5 11 7 8 9 6 10
Partition around it.
0 4 3 2 1 5 11 7 8 9 6 10
Recurse on both subarrays.
Choose a pivot and partition around it.
0 1 2 4 3 5 6 7 11 8 9 10
Choose a pivot and partition around it.
Recurse on both subarrays.
0 1 2 4 3 5 6 7 11 8 9 10
Recurse on both subarrays.

Quicksort

```
void quickSort(int array[], int l, int r) {
  if (l < r) {
    int pivot = array[r];
    int pos = partition(array, l, r, pivot);

    quickSort(array, l, pos - 1);
    quickSort(array, pos + 1, r);
  }
}
```

Quicksort

```
// It searches for x in arr[l..r], and partitions the array around x.
int partition(int arr[], int l, int r, int x)
{
    // Search for x in arr[l..r] and move it to end
    int i;
    for (i=l; i<r; i++)
        if (arr[i] == x)
           break;
    swap(&arr[i], &arr[r]);

    // Standard partition algorithm
    i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}
```

Quicksort

```
void quickSort(int array[], int l, int r) {
  if (l < r) {
    int pivot = array[r];
    int pos = partition(array, l, r, pivot);

    quickSort(array, l, pos - 1);
    quickSort(array, pos + 1, r);
  }
}
```

Worst-case runtime

$O(n^2)$

Randomized Quicksort

```
void quickSort(int array[], int l, int r) {
  if (l < r) {
    int pivot = array[l + rand() % n];
    int pos = partition(array, l, r, pivot);

    quickSort(array, l, pos - 1);
    quickSort(array, pos + 1, r);
  }
}
```

Worst-case

$O(n^2)$

Think of this as the adversary chooses the randomness.

Expected

$O(n \log(n))$

Initial Observations

There's a really good case, in which partition always picks the median element as the pivot.

What's the recurrence relation? 🤔

Initial Observations

There's a really good case, in which partition always picks the median element as the pivot.

What's the recurrence relation? 🤔

T(0) = T(1) = Θ(1)

$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$ Runtime of `partition`.

= `O(nlogn)` Master method a = 1, b = 2, d = 1.

Initial Observations

There's a really good case, in which partition always picks the median element as the pivot.

What's the recurrence relation? 🤔

T(0) = T(1) = Θ(1)

$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$ Runtime of `partition`.

= `O(nlogn)` Master method a = 1, b = 2, d = 1.

There's a really bad case, in which partition always picks the smallest or largest element as the pivot.

What's the recurrence relation? 🤔

Initial Observations

There's a really good case, in which partition always picks the median element as the pivot.

What's the recurrence relation? 🤔

$T(0) = T(1) = \Theta(1)$

$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$ Runtime of `partition`.

$= O(n\log n)$ Master method a = 1, b = 2, d = 1.

There's a really bad case, in which partition always picks the smallest or largest element as the pivot.

What's the recurrence relation? 🤔

$T(0) = T(1) = \Theta(1)$

$T(n) = T(n-1) + \Theta(n)$

$= O(n^2)$ Iteration method

Expected Runtime of Randomized Quicksort

the expected runtime of `quicksort` is `O(nlogn)`

We can prove it through counting the number of times two elements get compared!

- This might not seem intuitive at first, but it's an approach you can use to analyze runtime of randomized algorithms.

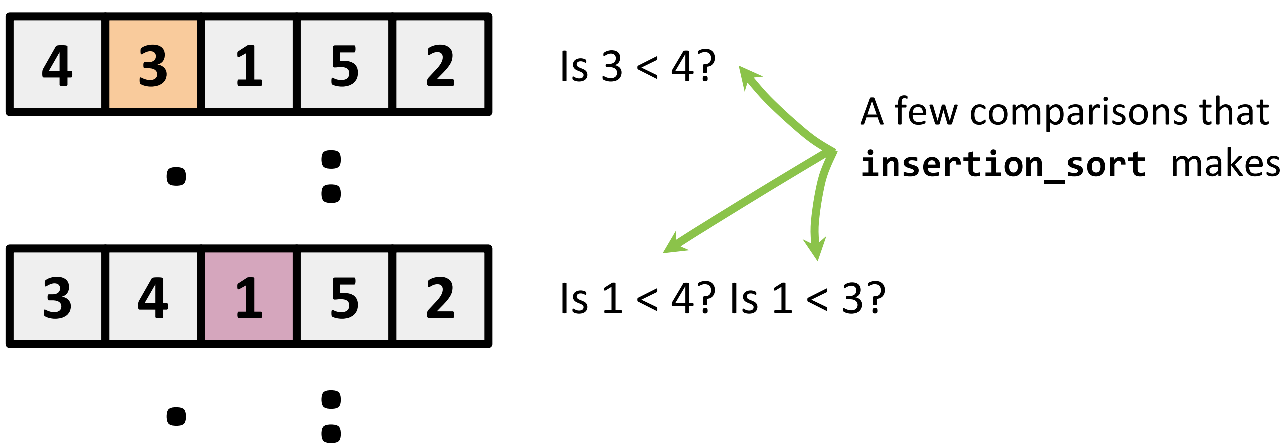
Lower-bound of Comparison-based Sorting

Sorting

- We've seen a few sorting algorithms
  - Insertion sort is worst-case **$O(n^2)$**-time.
  - Mergesort is worst-case **$O(n \log(n))$**-time.
- Can we do better?

Comparison-Based Sorting

- Comparison-based algorithms use “comparisons” to achieve their output.
  - **insertion_sort** and **merge_sort** are comparison-based sorting algorithms.
  - Linear-time **select** is a comparison-based algorithm.
  - Later, we’ll see a randomized comparison-based sorting algorithm called **quick_sort**.

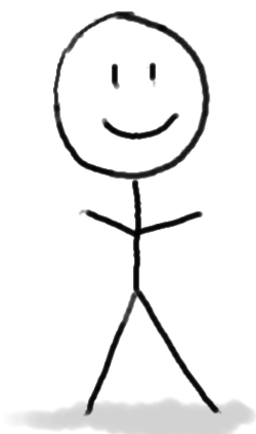


Comparison-Based Sorting

- Suppose we want to sort three items

Sort these three things.

Is  bigger than  ?



Algorithm

YES

The algorithm's job is to output a correctly sorted list of all the objects.

There is a genie who knows what the right order is.

The genie can answer YES/NO questions of the form:
is [this] bigger than [that]?

Comparison-Based Sorting

- **Theorem** [Lower bound of $\Omega(n \log(n))$]:

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

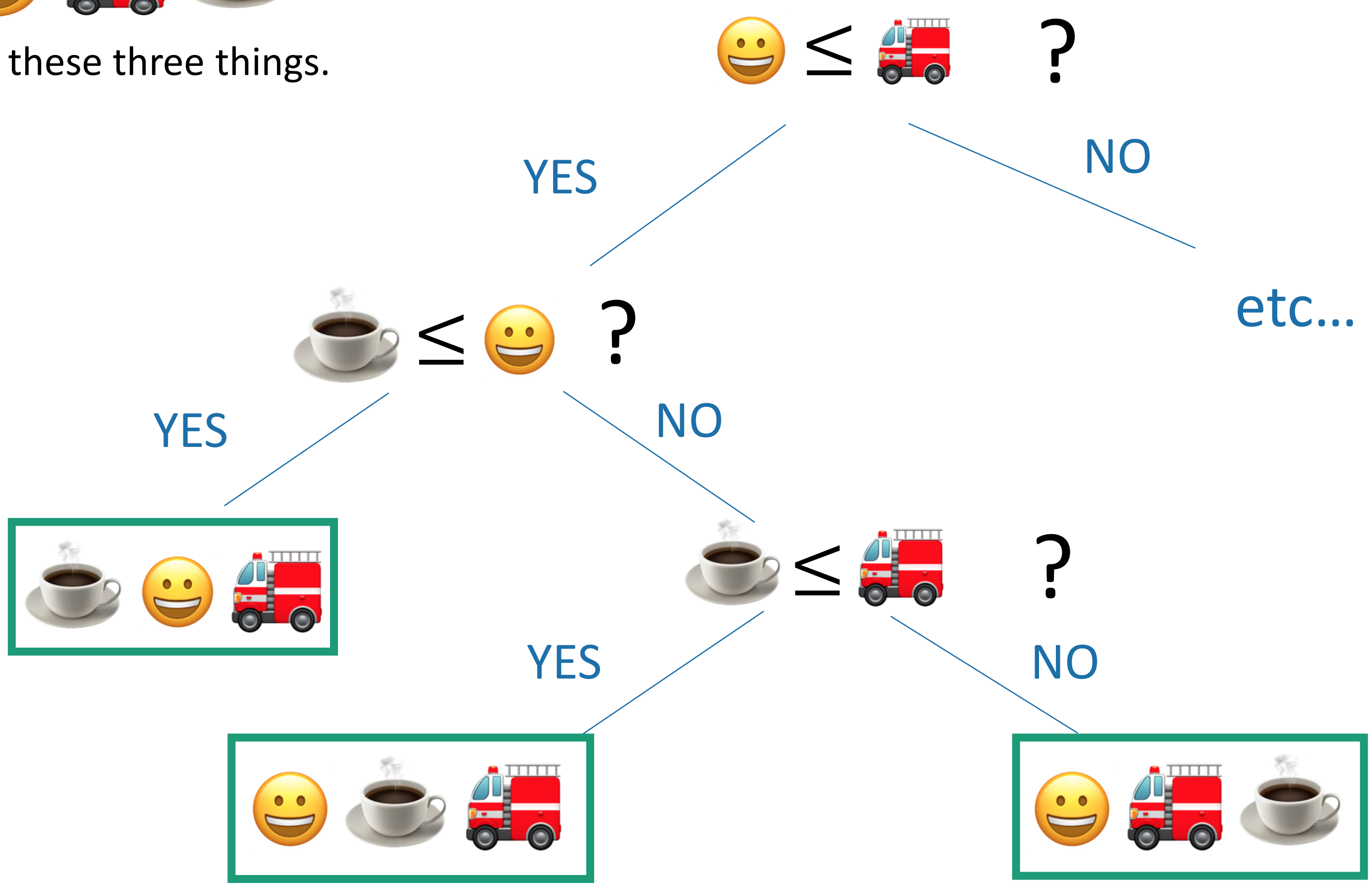
- How to prove this?
    1. Consider all comparison-based algorithms, one-by-one, and analyze them.
    2. Don't do that.

Instead, argue that all comparison-based sorting algorithms produce a **decision tree**.
Then analyze decision trees.

Decision Tree

- Represent all comparisons as a **decision tree**

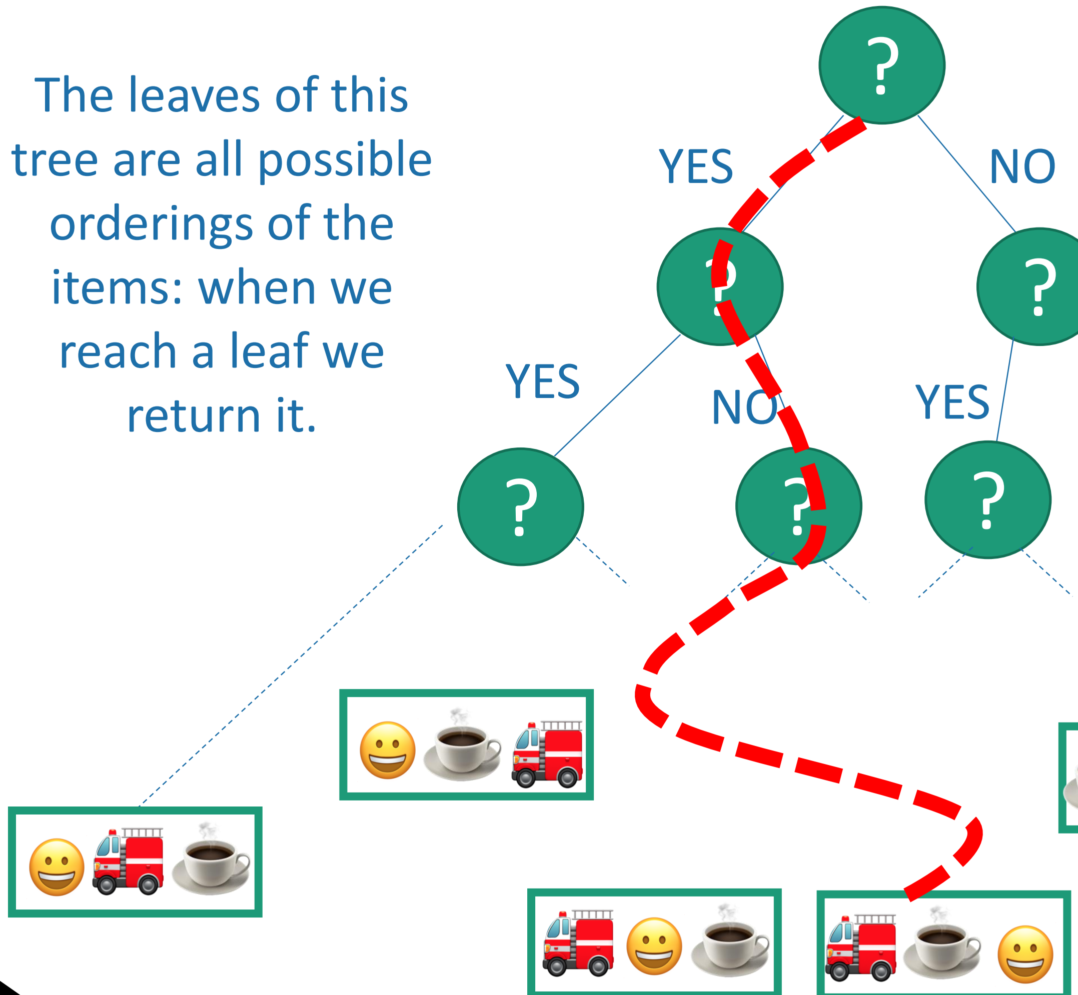
Sort these three things.



Decision Tree

- All comparison-based algorithms have an associated decision tree

The leaves of this tree are all possible orderings of the items: when we reach a leaf we return it.



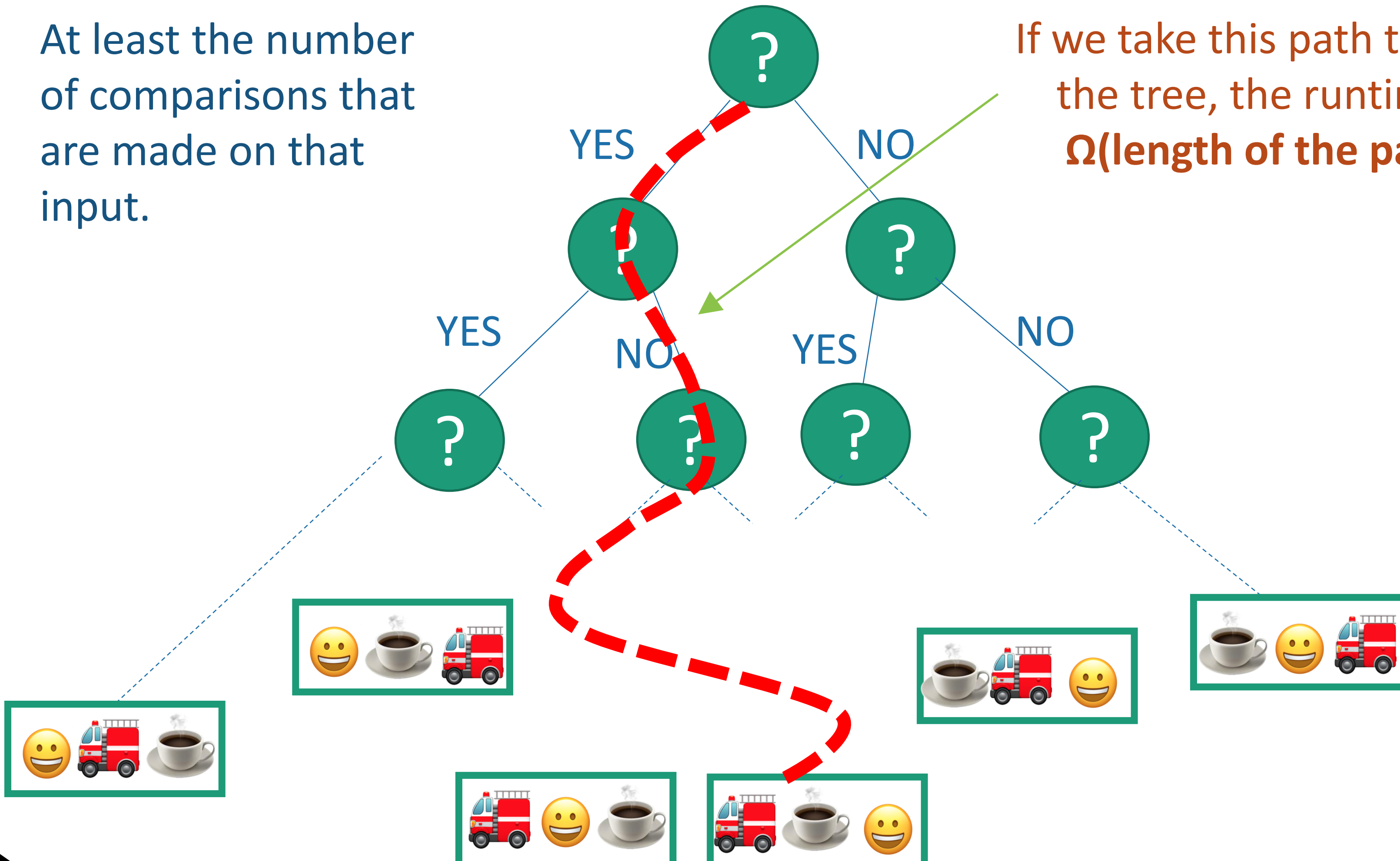
Running the algorithm on a given input corresponds to taking a particular path through the tree.

Decision Tree

- Q. What is the runtime on a particular input?

At least the number of comparisons that are made on that input.

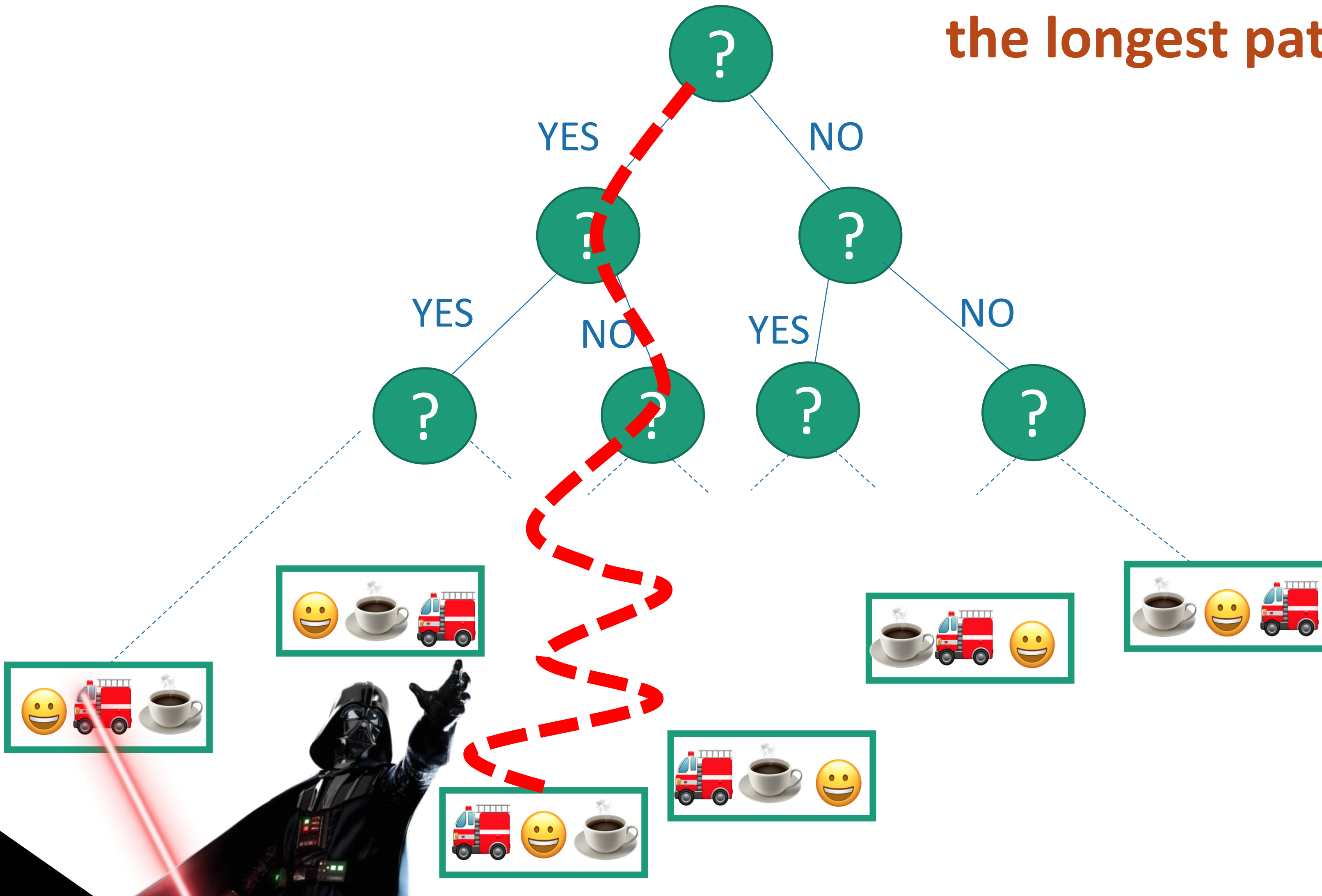
If we take this path through the tree, the runtime is **Ω(length of the path).**



Decision Tree

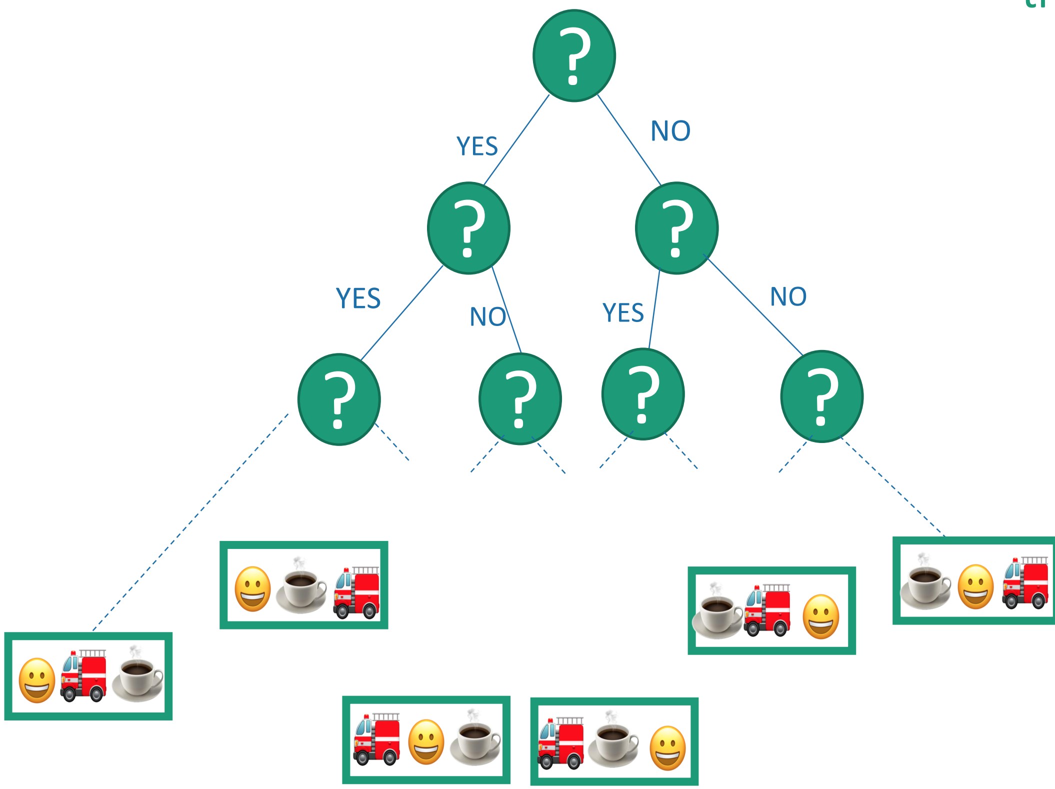
- Q. What is the worst-case runtime?

At least Ω(length of the longest path)



Decision Tree

- Q. How long is the longest path?



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

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

- log(n!) is about n log(n/e) = Ω(n log(n)).

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

Comparison-Based Sorting

- **Theorem** [Lower bound of $\Omega(n \log(n))$]:

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

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

Linear-Time Sorting

Is Linear-Time Sorting Nonsense?

- If any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$-time, then what's this nonsense about linear-time sorting algorithms?
  - We can achieve O(n) worst-case runtime if we make assumptions about the input.
  - e.g. They are integers ranging from 0 to k-1.

- Beyond comparison-based sorting algorithms
  - **Counting sort, Bucket sort, Radix sort**

Counting Sort

```
void countingsort(int array[], int n) {
    int count [k+1];          /* k is a MACRO constant */
    for (int i=0; i<k+1; i++)
        count[i] = 0;

    for (int i=0; i<n; i++)
        count[array[i]]++;

    for (int i=0, j=0; i<=k; i++)
    {
        while(count[i]>0)
        {
          array[j] = i;
          j++;
          count[i]--;
        }
     }
}
```

<u>**Worst-case**</u> **runtime** **O(n+k)**

Counting Sort

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

0	0	3	1	1	3	1	0

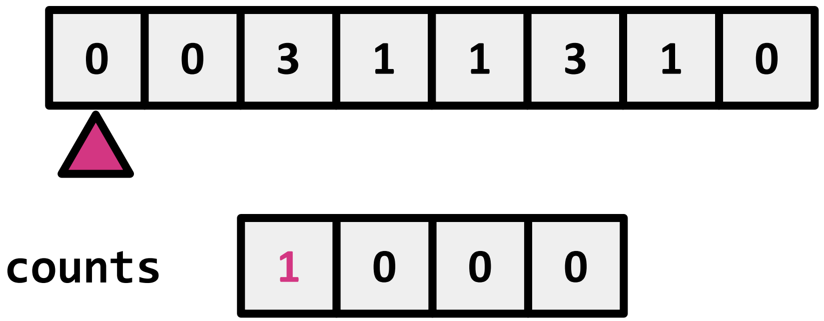
counts

0	0	0	0

```
void countingsort(int array[], int n) {
    int count [k+1];
    for (int i=0; i<k+1; i++)
        count[i] = 0;

    for (int i=0; i<n; i++)
        count[array[i]]++;
```

Counting Sort

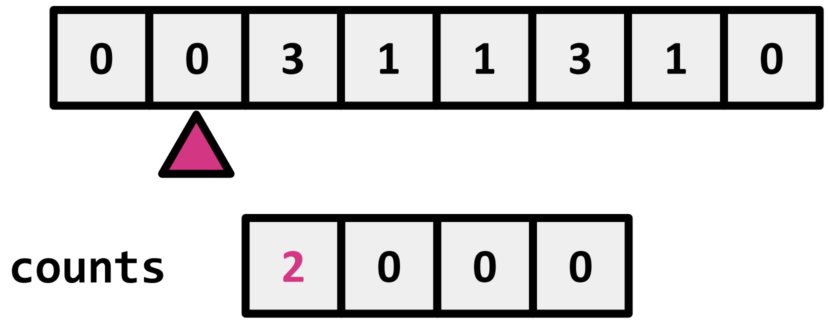
- Suppose **A** consists of 8 integers ranging from 0 to 3




```
void countingsort(int array[], int n) {
    int count [k+1];
    for (int i=0; i<k+1; i++)
        count[i] = 0;

    for (int i=0; i<n; i++)
        count[array[i]]++;
```

Counting Sort

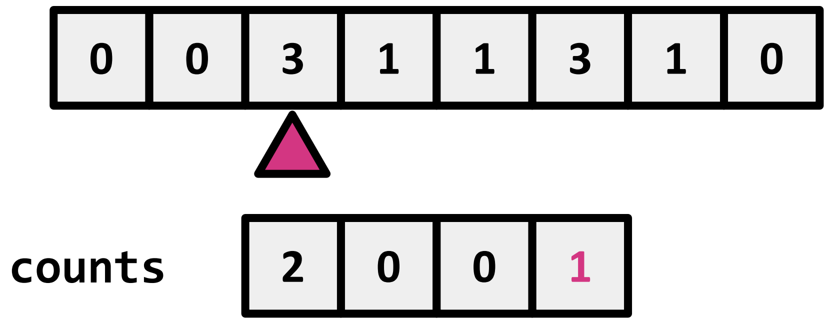
- Suppose **A** consists of 8 integers ranging from 0 to 3




```
void countingsort(int array[], int n) {
    int count [k+1];
    for (int i=0; i<k+1; i++)
        count[i] = 0;

    for (int i=0; i<n; i++)
        count[array[i]]++;
```

Counting Sort

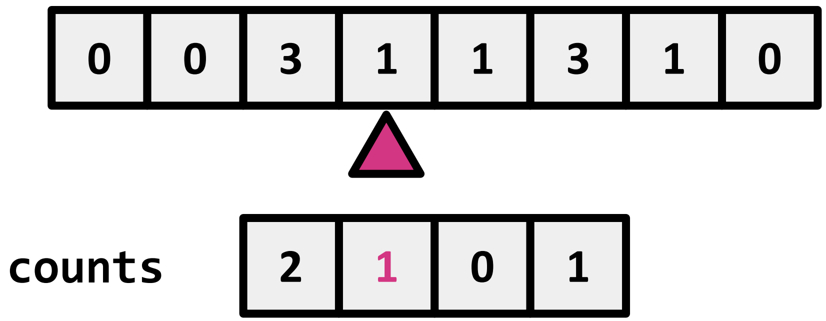
- Suppose **A** consists of 8 integers ranging from 0 to 3




```
void countingsort(int array[], int n) {
    int count [k+1];
    for (int i=0; i<k+1; i++)
        count[i] = 0;

    for (int i=0; i<n; i++)
        count[array[i]]++;
```

Counting Sort

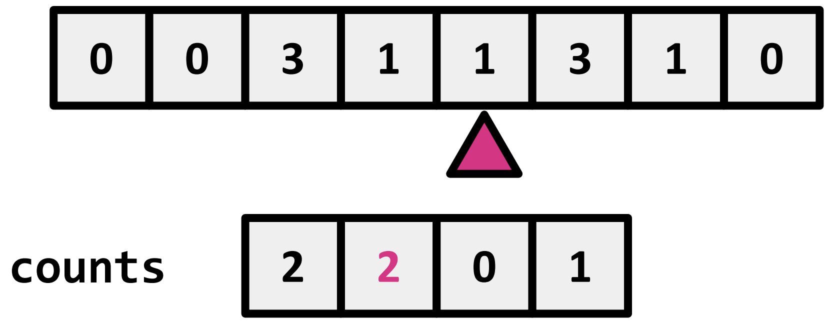
- Suppose **A** consists of 8 integers ranging from 0 to 3




```
void countingsort(int array[], int n) {
    int count [k+1];
    for (int i=0; i<k+1; i++)
        count[i] = 0;

    for (int i=0; i<n; i++)
        count[array[i]]++;
```

Counting Sort

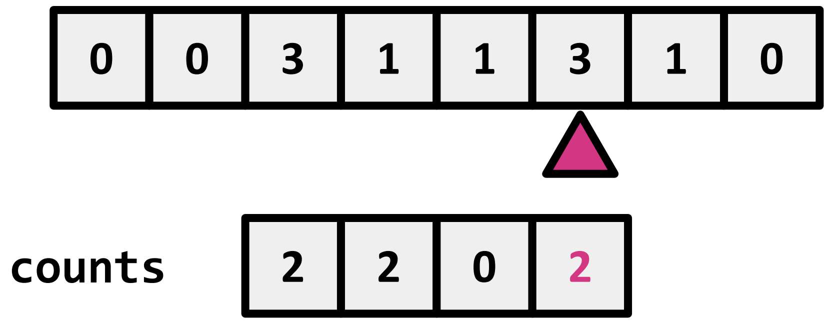
- Suppose **A** consists of 8 integers ranging from 0 to 3




```
void countingsort(int array[], int n) {
    int count [k+1];
    for (int i=0; i<k+1; i++)
        count[i] = 0;

    for (int i=0; i<n; i++)
        count[array[i]]++;
```

Counting Sort

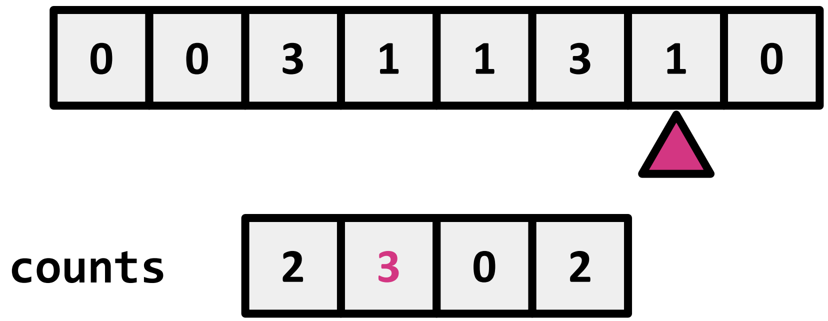
- Suppose **A** consists of 8 integers ranging from 0 to 3




```
void countingsort(int array[], int n) {
    int count [k+1];
    for (int i=0; i<k+1; i++)
        count[i] = 0;

    for (int i=0; i<n; i++)
        count[array[i]]++;
```

Counting Sort

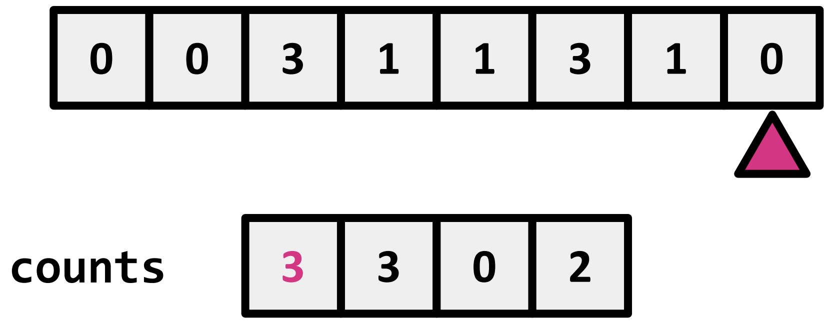
- Suppose **A** consists of 8 integers ranging from 0 to 3




```
void countingsort(int array[], int n) {
    int count [k+1];
    for (int i=0; i<k+1; i++)
        count[i] = 0;

    for (int i=0; i<n; i++)
        count[array[i]]++;
```

Counting Sort

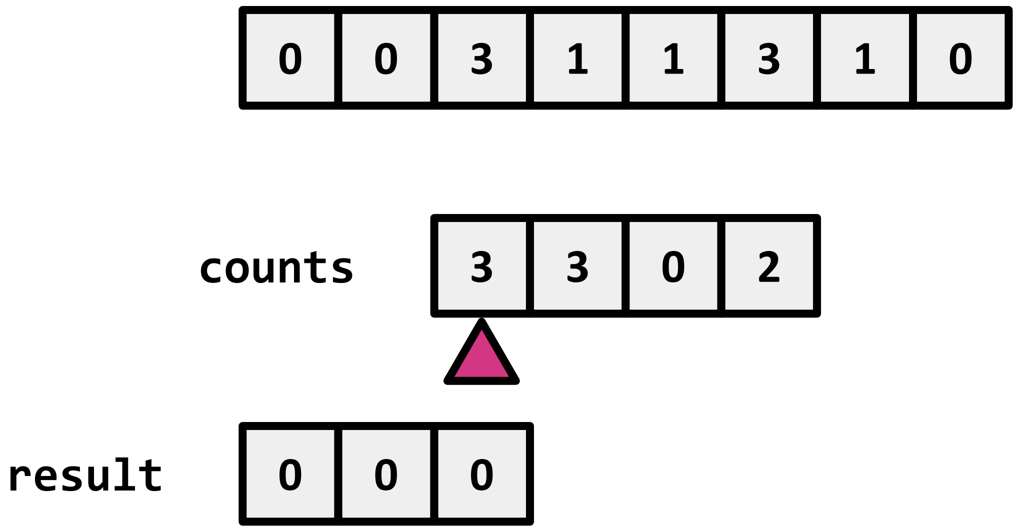
- Suppose **A** consists of 8 integers ranging from 0 to 3




```
void countingsort(int array[], int n) {
    int count [k+1];
    for (int i=0; i<k+1; i++)
        count[i] = 0;

    for (int i=0; i<n; i++)
        count[array[i]]++;
```

Counting Sort

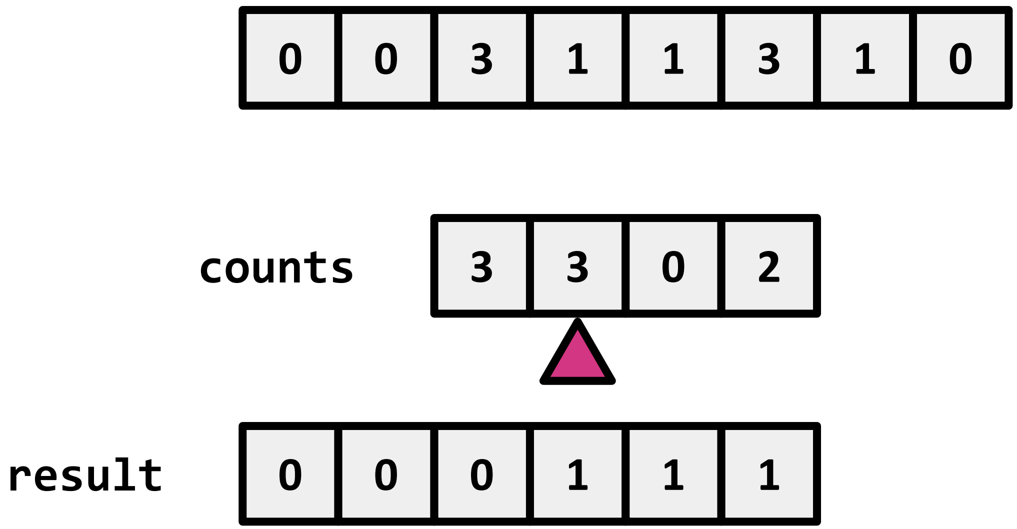
- Suppose **A** consists of 8 integers ranging from 0 to 3




```
 for (int i=0, j=0; i<=k; i++)
{
    while(count[i]>0)
    {
      array[j] = i;
      j++;
      count[i]--;
    }
 }
```

Counting Sort

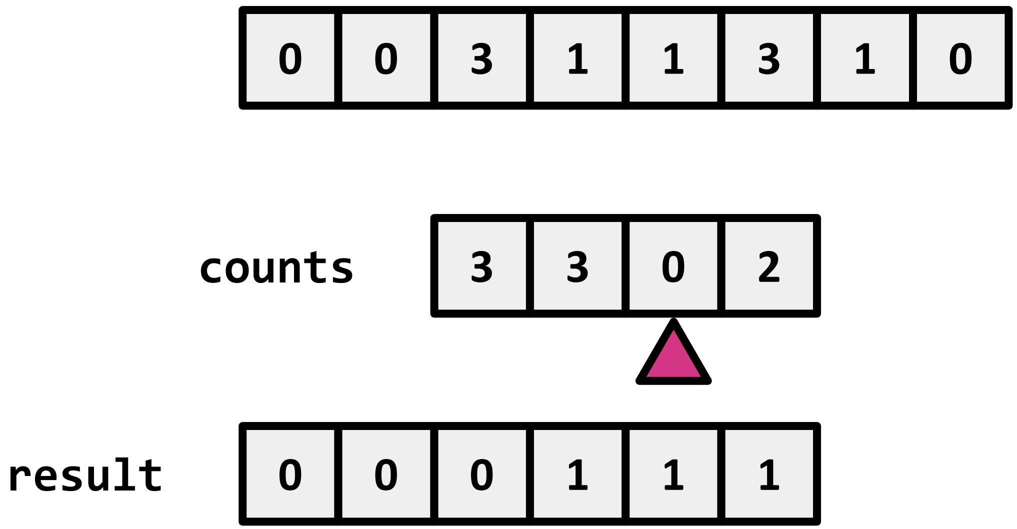
- Suppose **A** consists of 8 integers ranging from 0 to 3




```
for (int i=0, j=0; i<=k; i++)
{
    while(count[i]>0)
    {
      array[j] = i;
      j++;
      count[i]--;
    }
}
```

Counting Sort

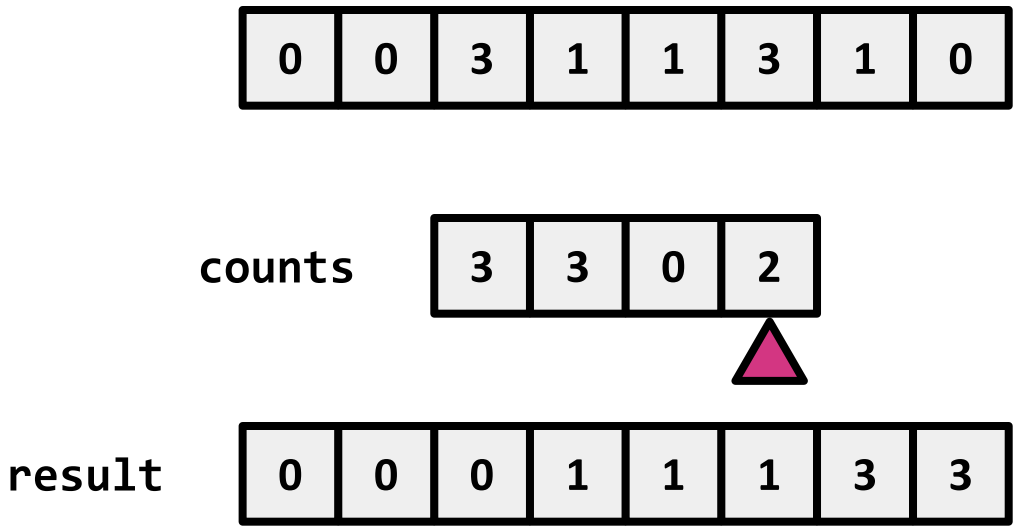
- Suppose **A** consists of 8 integers ranging from 0 to 3




```
for (int i=0, j=0; i<=k; i++)
{
    while(count[i]>0)
    {
      array[j] = i;
      j++;
      count[i]--;
    }
}
```

Counting Sort

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

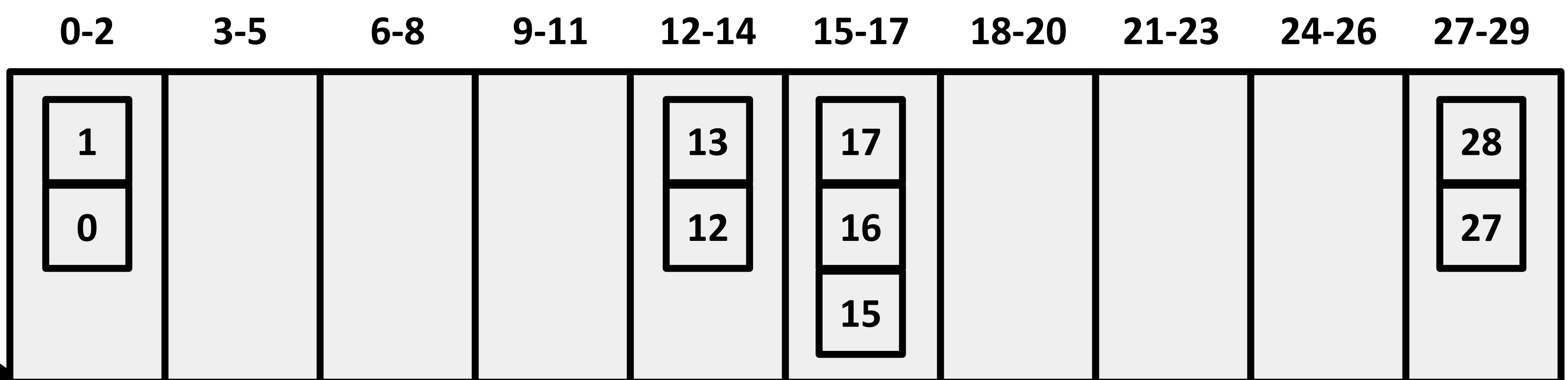



```
for (int i=0, j=0; i<=k; i++)
{
    while(count[i]>0)
    {
      array[j] = i;
      j++;
      count[i]--;
    }
}
```

Bucket Sort

Bucket Sort

- Bucket sort: similar to the counting sort, but
- Might be multiple keys per bucket, so buckets need another stable_sort to be sorted.



Bucket Sort

```
static void bucketsort(int Array[], int n, int k)
{
    //creating empty buckets. Suppose that bucket_number is a macro constant
    vector<int> bucket[bucket_number];

    //transfer elements of array into respective bucket
    for (int i = 0; i < n; i++)
    {
      int b = Array[i] / ceil (k / bucket_number);
      bucket[b].push_back(Array[i]);
    }

    //sort all elements of each bucket
    if (bucket_number < k)
    {
        for (int i = 0; i < bucket_number; i++)
            sort(bucket[i].begin(), bucket[i].end());
    }

    //combine all buckets to create sorted list
    int m = 0;
    for (int i = 0; i < bucket_number; i++)
    {
      for (int j = 0; j < bucket[i].size(); j++)
      {
         Array[m] = bucket[i][j];
         m++;
      }
    }
}
```

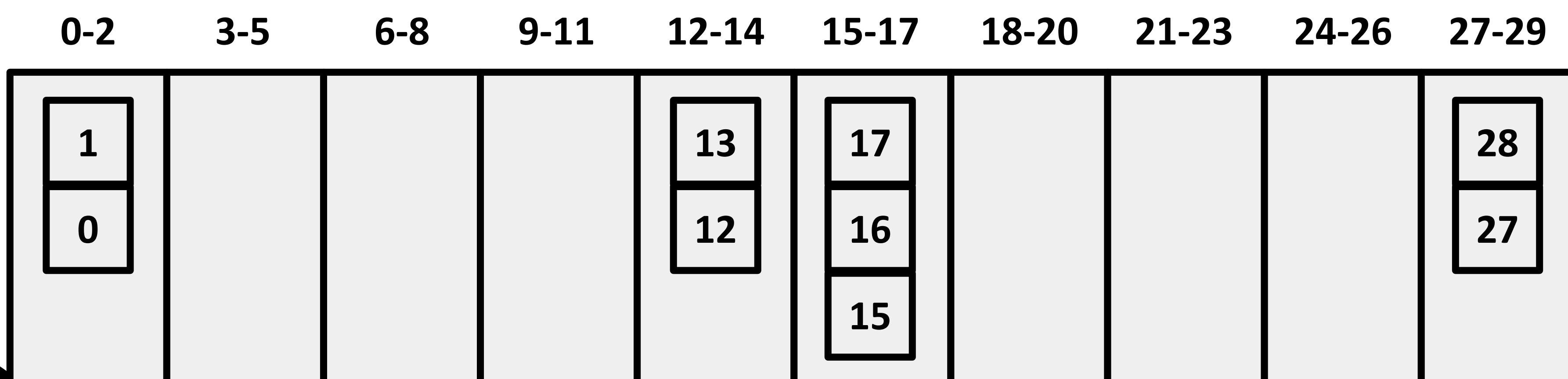
Bucket Sort

```
static void bucketsort(int Array[], int n, int k)
{
    //creating empty buckets. Suppose that bucket_number is a macro constant
    vector<int> bucket[bucket_number];

    //transfer elements of array into respective bucket
    for (int i = 0; i < n; i++)
    {
      int b = Array[i] / ceil (k / bucket_number);
      bucket[b].push_back(Array[i]);
    }

    //sort all elements of each bucket
    if (bucket_number < k)
    {
        for (int i = 0; i < bucket_number; i++)
            sort(bucket[i].begin(), bucket[i].end());
    }

    //combine all buckets to create sorted list
    int m = 0;
    for (int i = 0; i < bucket_number; i++)
    {
      for (int j = 0; j < bucket[i].size(); j++)
      {
         Array[m] = bucket[i][j];
         m++;
      }
    }
}
```

<u>Worst-case</u> runtime $O(\max\{n \log(n), n+k\})$

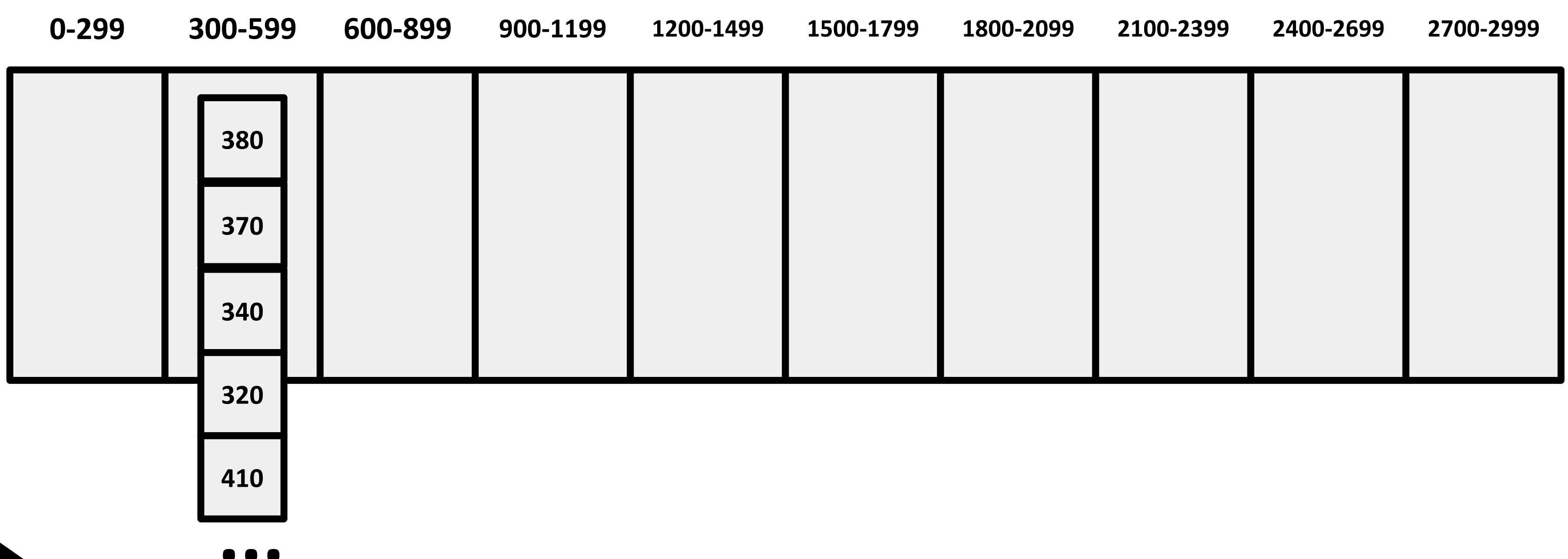
Bucket Sort

- Two cases for num_buckets and k:
  - **k ≤ num_buckets** At most one key per bucket, so buckets don't need another stable_sort to be sorted (similar to counting_sort).
  - **k > num_buckets** Might be multiple keys per bucket, so buckets need another 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:



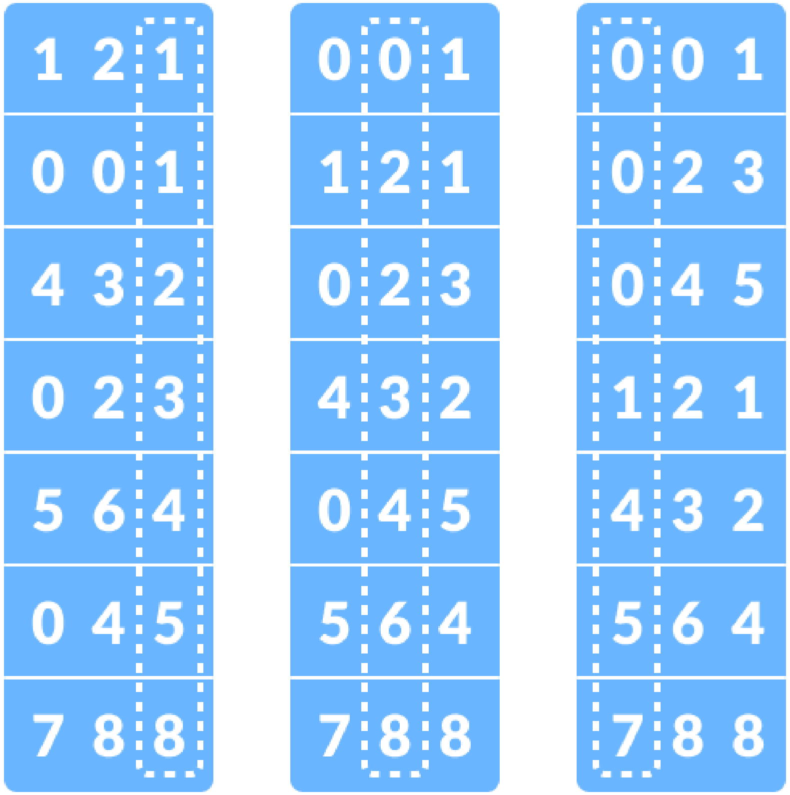
Bucket Sort

- In an extreme case, a bucket might receive all of the inserted keys.
- Suppose k = 3000 and num_buckets = 10.
  - 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.



Radix Sort

Radix Sort



sorting the integers according to units, tens and hundreds place digits

Radix Sort

```
void radixsort(int array[], int size) {
  // Get maximum element
  int max = getMax(array, size);

  // Apply counting sort to sort elements based on place value.
  for (int place = 1; max / place > 0; place *= 10)
    countingSort(array, size, place);
}
```

Radix Sort

```
void countingSort(int array[], int size, int place) {
  const int max = 10;
  int* output = new int[size];
  int* count = new int[max];

  for (int i = 0; i < max; ++i)
    count[i] = 0;

  // Calculate count of elements
  for (int i = 0; i < size; i++){
    key = (array[i] / place) % 10;
    count[key]++;
  }

  // Calculate cummulative count
  for (int i = 1; i < max; i++)
    count[i] += count[i - 1];

  // Place the elements in sorted order
  for (int i = size - 1; i >= 0; i--) {
    key = (array[i] / place) % 10;
    output[count[key] - 1] = array[i];
    count[key]--;
  }

  for (int i = 0; i < size; i++)
    array[i] = output[i];
}
```

Radix Sort

```
void radixsort(int array[], int size) {
  // Get maximum element
  int max = getMax(array, size);

  // Apply counting sort to sort elements based on place value.
  for (int place = 1; max / place > 0; place *= 10)
    countingSort(array, size, place);
}
```

<u>Worst-case</u> runtime **O(d(n+k))**

Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **`radix_sort(A,8)`**:

A	31	5	210	14	95	477	555	125

place	1

Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **`radix_sort(A,8)`**:

`A`

31	5	210	14	95	477	555	125

`place` 1

`A_new`

210	031	014	005	095	555	125	477

Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **`radix_sort(A,8)`**:

A	210	031	014	005	095	555	125	477

place	10

Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **`radix_sort(A,8)`**:

`A`

210	031	014	005	095	555	125	477

`place` 10

`A_new`

005	210	014	125	031	555	477	095

Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **`radix_sort(A,8)`**:

A	005	210	014	125	031	555	477	095

place	100

Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **`radix_sort(A,8)`**:

A							
005	210	014	125	031	555	477	095

place 100

A_new							
005	014	031	095	125	210	477	555

The story so far

- If we use a comparison-based sorting algorithm, it **MUST** run in time $\Omega(n\log(n))$
- If we assume a bit of structure on the values (small integers or other reasonable data), we have an $O(n)$-time sorting algorithm

Why would we ever use a comparison-based sorting algorithm??

Comparison-Based vs. Linear-Time Sorting

- Why would we ever use a comparison-based sorting algorithm?
  - **It has lots of precision…**

π	$\frac{1234}{9876}$	**e**	**37!**	**6.0221409**	n^n	**42**

Comparison-Based vs. Linear-Time Sorting

- Why would we ever use a comparison-based sorting algorithm?
  - **It has lots of precision…**

π	$\frac{1234}{9876}$	e	37!	6.0221409	n^n	42

We can compare these pretty quickly (just look at their most significant digit):

- π = 3.14159…
- e = 2.71818…

But **`radix_sort`** requires us to look at all digits, which is problematic—both have infinitely many!

Comparison-Based vs. Linear-Time Sorting

- Why would we ever use a comparison-based sorting algorithm?
  - **It has lots of precision…**

π	**$\frac{1234}{9876}$**	**e**	**37!**	**6.0221409**	**n^n**	**42**

We can compare these pretty quickly (just look at their most significant digit):

- π = 3.14159…
- e = 2.71818…

But **`radix_sort`** requires us to look at all digits, which is problematic—both have infinitely many!

Even with integers, if it's really big, **`radix_sort`** is slow.

Comparison-Based vs. Linear-Time Sorting

- Why would we ever use a comparison-based sorting algorithm?
  - **It has lots of precision...**

π	$\frac{1234}{9876}$	**e**	**37!**	**6.0221409**	n^n	**42**

We can compare these pretty quickly (just look at their most significant digit):
- π = 3.14159...
- e = 2.71818...

But **`radix_sort`** requires us to look at all digits, which is problematic—both have infinitely many!

Even with integers, if it's really big, **`radix_sort`** is slow.

  - **`radix_sort`** needs extra memory for the buckets (not in-place).
  - Need to know ordering and buckets ahead of time for linear-time sorting.

Today's Outline

- ~~Quicksort~~ **Done!**
- ~~Linear-Time Sorting~~
  - ~~Comparison-based sorting lower bounds~~ **Done!**
  - ~~*Algorithms: Counting sort, bucket sort, and radix sort*~~ **Done!**
  - Reading: CLRS 8.1-8.2