How fast can you sort?

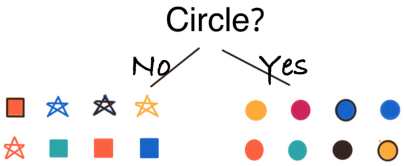
In this lecture, we tackle the issue of determining *how fast* we can sort n items. We have previously seen algorithms such as Mergesort and Heapsort that can sort in $\Theta(n \log n)$ time, even in the worst case. But how do we know there is not *another* algorithm out there that can sort asymptotically *faster*? Could there be a $O(n)$ algorithm? Or a $O(\log n)$ algorithm? Do researchers need to continue to search for even faster sorting algorithms? We begin this lecture by answering that exact question: we shall show that any comparison-based algorithm, whatever it may be, has a runtime of at least $\Omega(n \log n)$.

1 A Sorting Lower bound

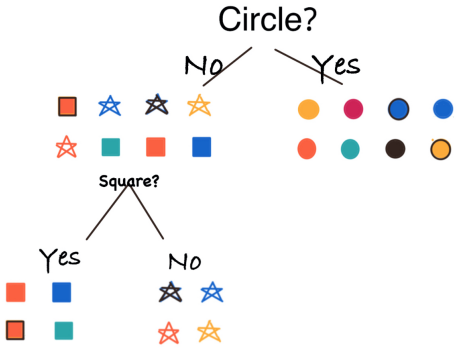
1.1 A decision tree

Let's play a guessing game. Suppose I draw the following objects, and I told you I was going to think of one of them in my head:

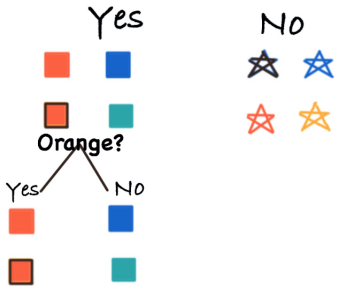
Your job is to figure out which one I'm thinking of - and you may ask me any **yes** or **no** question. What kind of question would you start out with? You may notice that there is only one red object, so does it seem reasonable to first ask "*Is it red?*". Of course you might get lucky if indeed my object was the red one, but if I answer no, then you still have 15 other items as possible correct answers. Instead, most of you would probably intuitively start off with a question that divides the set of possibilities in *half*. Since about half the objects are circles, you could ask "*Is your object a circle?*". By doing so, regardless of my answer, you will have reduced your possibilities to the following two cases:



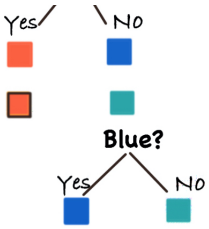
Suppose my answer was *no*, then your next question would be directed to all the objects on the left side above. Using the same approach, your next question could divide the possibilities into approximately two equal sizes. For example, you could ask "*Is the object a square?*".



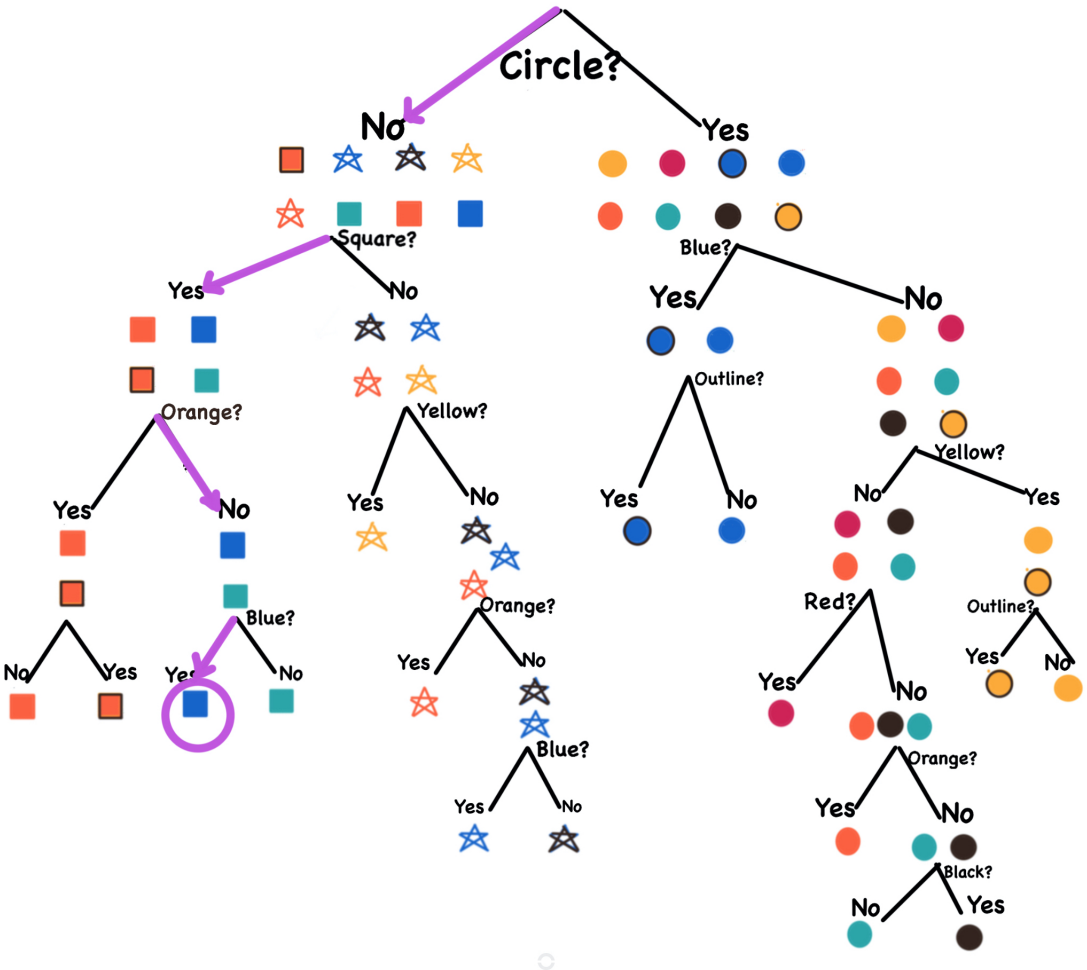
If my answer was *yes*, then again, you have only 4 possibilities remaining. Next you could ask if the object is orange,



and finally if I answered *no*, you could then ask if it is blue, and my answer would give you the final correct object - the blue square.



This process is an example of what is called a **decision tree**. The sequence of questions we used to solve this problem represents a *path* in the decision tree:



We now look at the specific properties of the decision tree:

- The possible outcomes (or answers) lie at the **leaves** of the tree. Depending on what answers I may have given, each **path** in the tree leads to a different answer.

- The **number of questions** that you need to ask before arriving at the answer is the **length of the path** from the root to the leaf. In the above example, there were four questions that were required in order to arrive at the correct answer.
- The **worst-case** number of questions required to arrive at an answer is the length of the **longest path** in the tree. The decision tree above has a path of length 6, and thus in some situations, we would require 6 questions to arrive at our answer.
- To **minimize** the number of questions that are needed in the worst-case, we minimize the **height** of the decision tree.

From the above, it is clear that we would like to consider decision trees with minimum height. Intuitively, that is what we were doing with our first question when we tried to split our outcomes into two equal-sized sets. In fact, if we were able to ask questions that always divided our possible answers into *two equal sizes*, then we would have a decision tree of minimum height. Such a tree is described below:

Height of the decision tree:

- If we have a binary tree with L leaves, the height is **at least** $\log_2 L$.
- If we have a decision tree with n possible outcomes, then the **minimum number of questions needed** is $\log_2 n$.

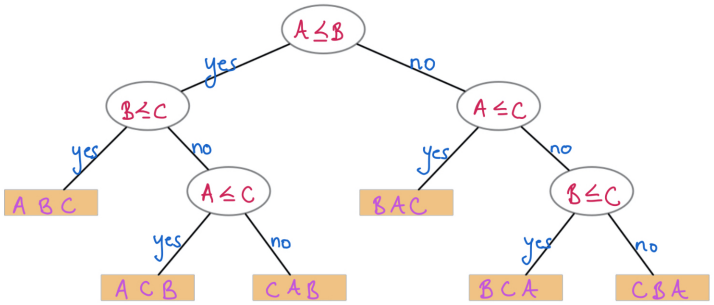
1.2 The decision tree model of comparison-based sorting

In this section we apply the previous result to obtain a **lower bound** for comparison-based sorting algorithms. A comparison-based sorting algorithm is one that uses only comparisons between pairs of elements in order to arrive at the "*answer*". For example, if the input is $\{3, 2, 5\}$, then the algorithm carries out comparisons between pairs of elements in order to arrive at the conclusion: $\{2, 3, 5\}$

Comparison-based sorting algorithms can be modelled using decision trees. Each comparison is simply a *yes/no* question, such as:

"Is the second number smaller than the first ?"

For example, suppose we wish to sort the elements $\{A, B, C\}$. We could begin by asking if $A \leq B$. If the answer were yes, then we could next ask if $B \leq C$. If again the answer were yes, then the conclusion is that the numbers are sorted as $\{A, B, C\}$. This represents just one particular path of the decision tree. Different paths would lead to different questions and results, and each path would result in a conclusion of how the elements are sorted:



For example, if the input were $\{A, B, C\} = \{5, 3, 1\}$, then we follow the path:
Question 1: Is $5 \leq 3$? No
Question 2: Is $5 \leq 1$? No
Question 3: Is $3 \leq 1$? No
Then the sorted order must be $\{C, B, A\} = \{1, 3, 5\}$.

The above is just one example of a decision tree that sorts three elements. Any comparison-based sorting algorithm can be modelled as a decision tree that uses yes/no comparisons, and they may all have different heights. The fundamental question that we need to answer is:

What is the number of comparisons required to sort the elements?

We learned in the above section that the maximum number of questions needed is based on the height of the decision tree. Let's see how this relates to our decision tree for sorting:

- A decision tree for sorting n elements will have at least $n!$ **possible outcomes** (one for each possible sorted order), and so it will have at least $n!$ leaves.
- The **minimum height** of the decision tree is then:

$$log_2(n!)$$

since we have $n!$ leaves.

The conclusion is that a decision-tree based on comparisons will have height at least $\log_2(n!)$. This means we need *at least* $\log_2(n!)$ comparisons to sort our data in the worst-case. With a big of math, one can show that $\log_2(n!)$ is in fact $\Omega(n \log n)$. This allows as to conclude the following very famous result:

Theorem 1. *Any comparison based sorting algorithm requires* $\Omega(n \log n)$ *comparisons in the worst-case*

How does this relate to the algorithms we have already seen for sorting? In fact, both Heapsort and Mergesort are comparison-based algorihtms: it is possible to model them with a decision tree using comparisons. This means that they *require* $\Omega(n \log n)$ comparisons. Recall that we showed that Heapsort and Mergesort run in time $O(n \log n)$. Now we know that in fact these algorithms are **optimal**, in the sense that *no other comparison based algorithm can sort faster* in the worst-case.

We now turn to a few *fast* sorting algorithms - those that run in time $O(n)$. Note that they are **not** comparison-based. They are able to sort faster simply because they make *certain assumptions* about the elements, allowing them to sort without using comparisons.

2 Linear time Sorting: Counting Sort

Counting sort can only be applied to sorting problems where the input is of a particular type. Furthermore, the algorithm requires extra storage in order to operate.

Counting Sort:

Assumption: The input is a set of n integers where each integer is **at least** 0 **and at most** k

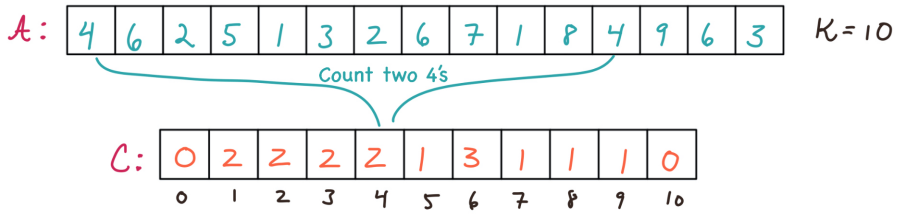
Additional storage: Counting sort uses an additional array indexed from 0 to k.

Runtime: $O(n + k)$ where k is the maximum integer in the input.

For example, if $k = 10$, then each number is an integer between 0 and 10.

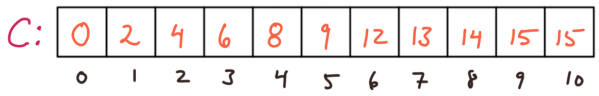
Counting sort works in a very simple way: for each number in the array, it *counts* the number of elements that are *less* than that number. For example, if you know that there are 7 numbers less than x, then you know that in the final sorted array, x must go in position 8. We go through the steps of the algorithm below. The original input is assumed to be in array A, and we initialize an *additional* array $C[0 \dots k]$.

- **Step 1:**. Count the number of occurrences of each element and place this in an array C. The value of $C[i]$ contains the number of occurrences of element i.



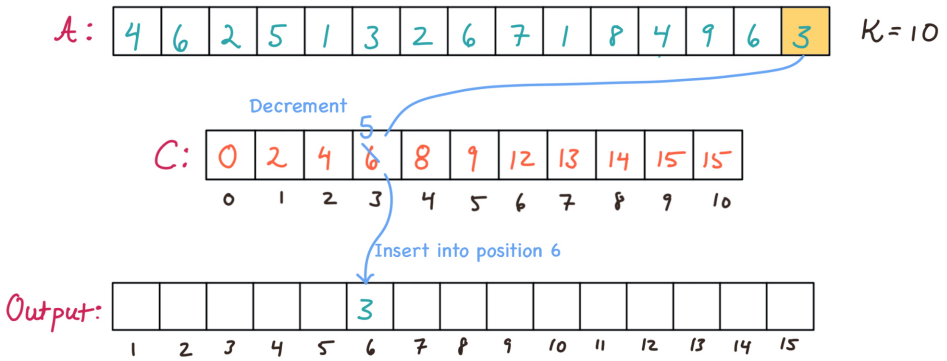
Time: The algorithm passes through the array A once, and for each element $A[i]$, the total count is updated in $C[A[i]]$. This takes time $O(n)$.

- **Step 2:** Update the array C so that $C[i]$ now contains the number of elements that are *less* than or equal to i. This can be accomplished with a simple loop left to right through array C, where we set $C[i] = C[i] + C[i-1]$



Time: This takes time $O(k)$ because we simply loop through the array C and update each element to the sum of the preceding values.

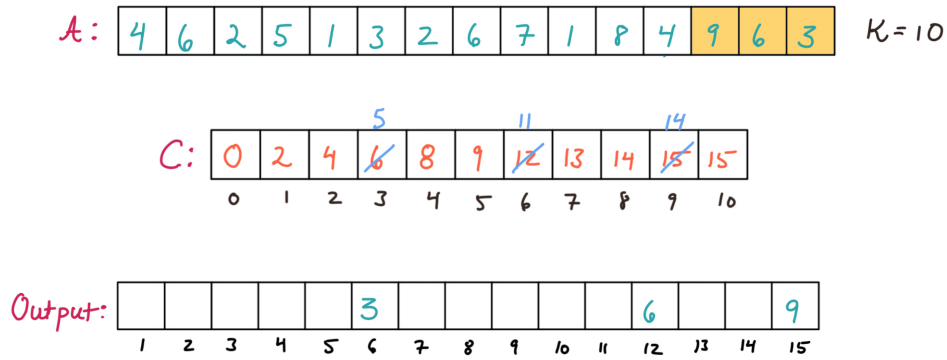
- **Step 3:** Now we can take advantage of the fact that $C[i]$ contains the number of elements that are less or equal to i. Starting at the back of array A, the algorithm processes each element $A[i]$ by looking up in C how many elements are *less than* $A[i]$. The element $A[i]$ is placed in its final sorted position in the output array according to how many elements are less than $A[i]$.



Note that the algorithm allows for duplicates. As each element $A[i]$ is positioned into the output array, we *decrease* the total number of elements *less than* $A[i]$ in the array C.

Time: This third step also runs in time $O(n)$ since it loops through the array A one time.

Here is a snapshot of the arrays after 2 more inserts into the output array:



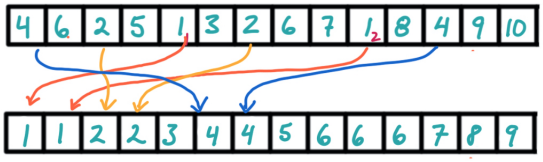
Running time:

The runtime of the above three steps is $O(n+k)$. If k is some multiple of n, the total runtime is $O(n)$.

Counting sort beats the lower bound we established for comparison-based sorting algorithms. This is because counting sort does *not* use comparisons to determine the final sorted order of the elements. Instead, it uses extra storage of size $O(k)$ to organize the elements in their sorted order. In normal comparison-based algorithms this is not possible since we have no maximum value for the elements.

2.1 A Stable Sort

Counting sort is an example of a *stable sort*. If there are duplicate items in the original array A, then those elements are placed in the final output array in the same order that they appeared in A. For example, our original input array A is shown below, and directly underneath is the final sorted array. Note that the original order of duplicates is maintained.



3 Linear time Sorting: Radix Sort

In this section we look at another sorting algorithm that beats the $\Omega(n \log n)$ lower bound. **Radix sort** is a sorting algorithm that actually uses counting sort as a sub-step. The assumptions are given below:

Radix Sort:

Assumption: each number is assumed to have at most **d digits**, and each digit has one of **r types**

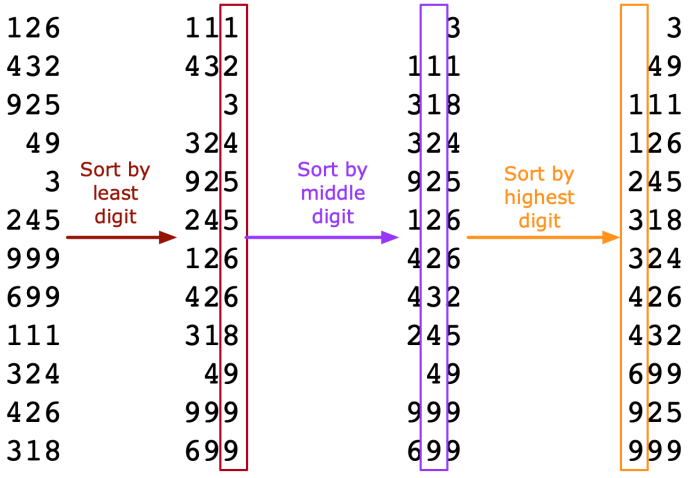
Additional storage: Additional array of size $O(r)$

Runtime: O(d(n+r)

For example, if we sort a set of numbers less than 1000, then each number has at most 3 digits, so $d = 3$. In the usual decimal system, each digit has 10 possibilities, and so $r = 10$.

Radix sort simply sorts the numbers *one digit* at a time, starting with the least significant digit. Counting sort is used for each pass of the algorithm, which means that an additional array of size $O(r)$ is needed. We demonstrate Radix sort by working through the following example where $d = 3$ and $r = 10$.

- Use counting sort to sort the numbers by the **least significant digit**. Repeat this process moving one digit to the left at each pass:



After 3 iterations, the final list of numbers is sorted. Why does this work? The answer lies in the fact that counting sort is a *stable sort*. As the numbers are sorted by digit, we do not "disrupt" the sorted order of the previous digit. For example, as the numbers above were sorted in the final pass, the relative ordering of the previous digits was not disrupted. In particular, 426 and 432 were correctly sorted in the purple column. On the last pass, since the digit 4 was repeated for both of those numbers, counting sort placed them in their original order: $426, 432$ in the final output.

Total runtime of Radix Sort:

Each pass of Radix sort requires Counting sort. Since there are exactly r possible digits, and the input size is n, the runtime of each pass of Counting sort is $O(n+r)$. Radix sort carries out exactly **d** iterations of Counting sort. Since each iteration of counting sort runs in time $O(n+r)$, the total runtime of Radix sort is $O(d(n+r))$. In most cases, the number of digits is a constant (i.e., d is a constant) and usually the number of possible values of each digit is a constant (i.e., r is a constant). Under these assumptions $O(d(n+r))$ is $O(n)$.

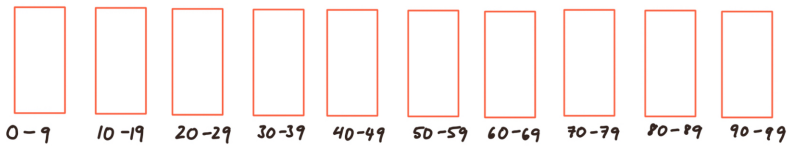
4 Linear time Sorting: Bucket Sort

We complete this lecture with one last linear time sorting algorithm. Unlike the previous two algorithms which *assume* something about the maximum size of each number, bucket sort can be carried out on any type of input array, and in fact, those numbers do not necessarily need to be integers.

We give an overview of how Bucket Sort works with the following example. Suppose we would like to sort the following n numbers:

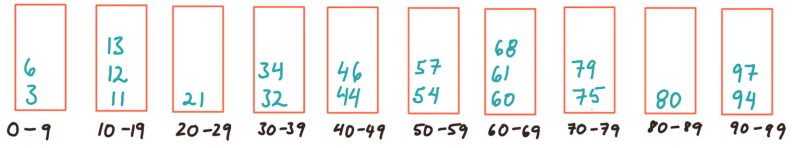
$$46, 32, 6, 79, 21, 11, 3, 44, 80, 94, 12, 13, 57, 61, 75, 68, 97, 34, 54, 60$$

- Create a set of empty *"buckets"*, where each bucket represents a possible **range**



- Pass through the elements of the input array, and put each number into the appropriate bucket. This step takes $O(n)$ time.

- Sort the elements in each bucket (using Insertion sort or any other comparison-based algorithm). This takes time $\sum_i O(n_i^2)$ where n_i is the number of items in bucket i.



- Go through the buckets in order and output the elements in sorted order. This takes time $O(n)$.

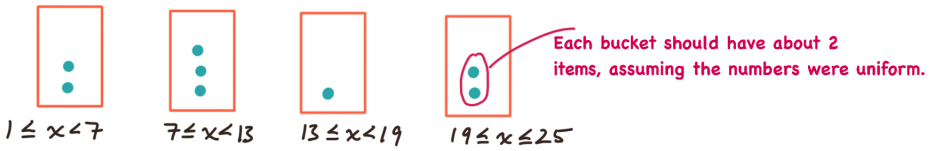
$$3, 6, 11, 12, 13, 21, 32, 34, 44, 46, 54, 57, 60, 61, 68, 75, 79, 80, 94, 97$$

How do we decide how many buckets to use? What is the best range for each bucket? The more items that fall in each bucket, the more numbers must be sorted by insertion sort. In fact, suppose *all* n numbers fell into *one* bucket. Then insertion sort would take $O(n^2)$ to sort that single bucket, meaning bucket sort would take $O(n^2)$ in the worst case.

Ideally, we want very few items in each bucket so that each bucket is sorted quickly - hopefully in constant time. Let's look first at a very natural solution. If the range of the n numbers is from 0 to M, then we could create n equal-size buckets, each of size M/n. For example, given $n = 5$ numbers ranging from 0 to 10, we could create 5 buckets each of size $10/5 = 2$. Intuitively, this may lead us to expect about *one* number in each box. This intuition would be completely valid if the numbers were *distributed uniformly* in the range 0 to 10, meaning if each number was equally likely to fall in any one bucket. In the next section, we show that this is indeed true: if the numbers are uniformly distributed over some range, we expect a **constant number** of elements in each bucket - leading to an overall runtime of Bucket Sort of $O(n)$.

4.1 Average-case Runtime for uniformly distributed data

Suppose our numbers are *real* numbers (not simply integers). Real numbers that are *uniformly distributed* over some range $a \leq x \leq b$, are those that are **equally likely** to fall into any bucket of size $(b-a)/k$. For example, a uniformly distributed number in $1 \leq x \leq 25$ is **equally likely** to fall into any bucket, as long as the buckets have equal size. Exactly how *many* we expect to fall into each bucket depends on how many buckets we have and how many numbers we have. Suppose we have 8 numbers uniformly distributed in the range $1 \leq x \leq 25$, and 4 buckets, then we expect on average about 2 numbers per bucket:



In this section we analyze the runtime of Bucket Sort in the specific case where:

1. The numbers are uniformly distributed in the range $0 \leq x \leq 1$.
2. The interval $0 \leq x \leq 1$ is divided into n equally-sized buckets.

Let n_i be the number of items that fall into bucket i. The exact number of items that fall into each bucket is actually *random*: this is the essence behind our assumption that the numbers themselves are selected randomly from the range $0 \leq x \leq 1$. We are interested in how many elements are in each bucket *on average* assuming we have this random input. The goal then is to determine the average (or expected value) of the runtime of insertion sort on each bucket: $O(n_i^2)$. In order to answer this question, we require the following facts:

Fact 1: Since each number is equally like to fall into any bucket, and there are n buckets, we expect about **one** number per bucket. Formally, this is called a *binomial random variable*.

Fact 2: For a binomial random variable where the expected value is a constant, the *square* of that value is *also* a constant. This means that n_i^2 is expected to be a constant. Simply said, the expected time it will take for insertion sort to sort each bucket is a **constant**.

We can now conclude with the runtime of Bucket Sort in this case. Bucket sort consists of:

- The time to distribute the numbers into the appropriate buckets: $\Theta(n)$
- The time to sort each bucket with insertion sort: $\sum_i O(n_i^2)$. Each $O(n_i^2)$ is constant, and therefore $\sum_i O(n_i^2) = O(n)$
- The time to go through each bucket and output the numbers in sorted order: $\Theta(n)$

In summary then, the expected runtime of bucket sort is $\Theta(n)$ when we use n buckets and the data is uniformly distributed in the range $0 \leq x \leq 1$.

Bucket Sort:

Assumption: The input is **uniformly distributed** over some range $a \leq x \leq b$. The numbers may be real numbers.

Runtime: The **expected** runtime depends on the number of buckets used. For n equally-sized buckets, the **expected** runtime is $\Theta(n)$.