CS5800: Algorithms

Week 3 – Linear Sorting & Selection

Dr. Ryan Rad

Spring 2025

Northeastern University

Linear Sorting & Selection

- Lower Bound for Sorting
- Counting Sort
- Bucket Sort
- Min & Max
- Order Statistics
- Class Activity
- HW2 Review

Last Week…

Let's solve problem P:

```
if (P is small enough)
   solution = DirectSolve(P)
else
   <P1,P2,…,Pk> = DivideProblem(P)
   for (i=1 to k)
      solutioni = Solve(Pi)
   solution = Combine(solution1,…,solutionk)
return solution
```

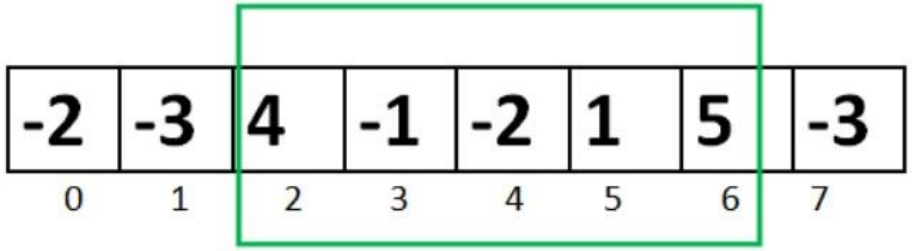
Or, use brute-force if (sub)problem is simple.

Divide the problem into smaller subproblems.

Recursively solve subproblems.

Combine solutions of subproblems to get solution for original problem.

Last Week...

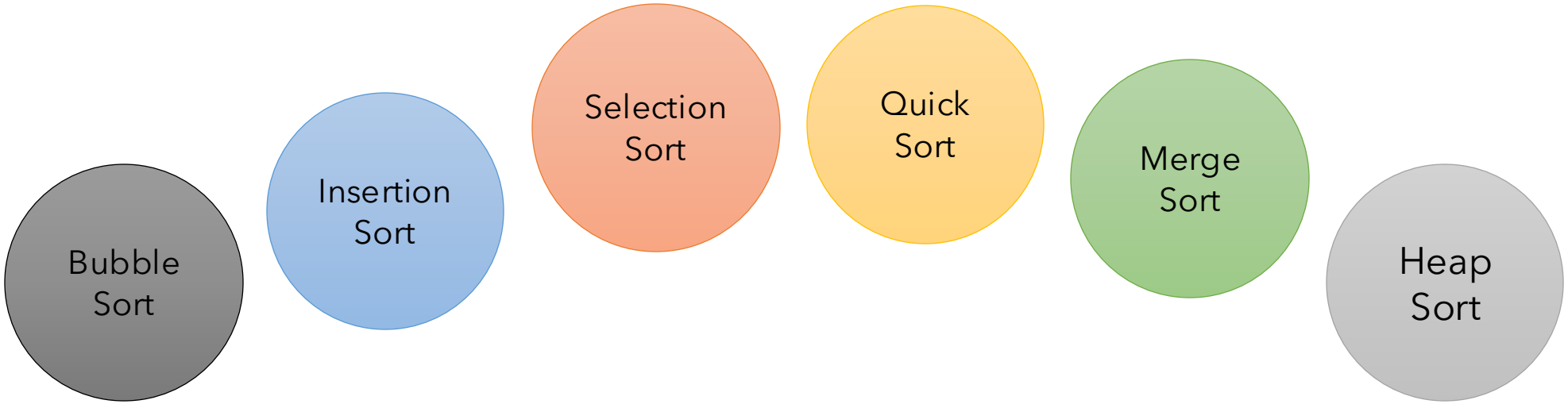


LeetCode

#53 Maximum Subarray

Medium

Lower Bounds For Sorting



Question:

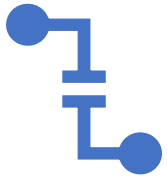
Which sorting algorithm is employed by several programming languages, including C++ (std::sort()), Java, and Python (sorted()) as default?

Q2 – Hybrid Sort (AKA, Introsort or Introspective Sort)

Key Points:

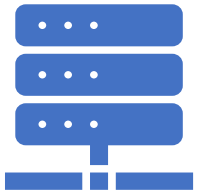
Hybrid algorithm combining Quicksort, Heapsort, and Insertion Sort

Many programming languages, including C++ (std::sort()), Java, and Python (sorted()), use Introsort or a variant of it as their default sorting algorithm for arrays or lists.



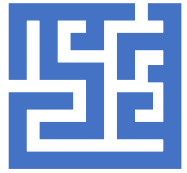
Efficiency:

Introsort typically outperforms quicksort in the worst-case scenario by switching to heapsort.



Versatility:

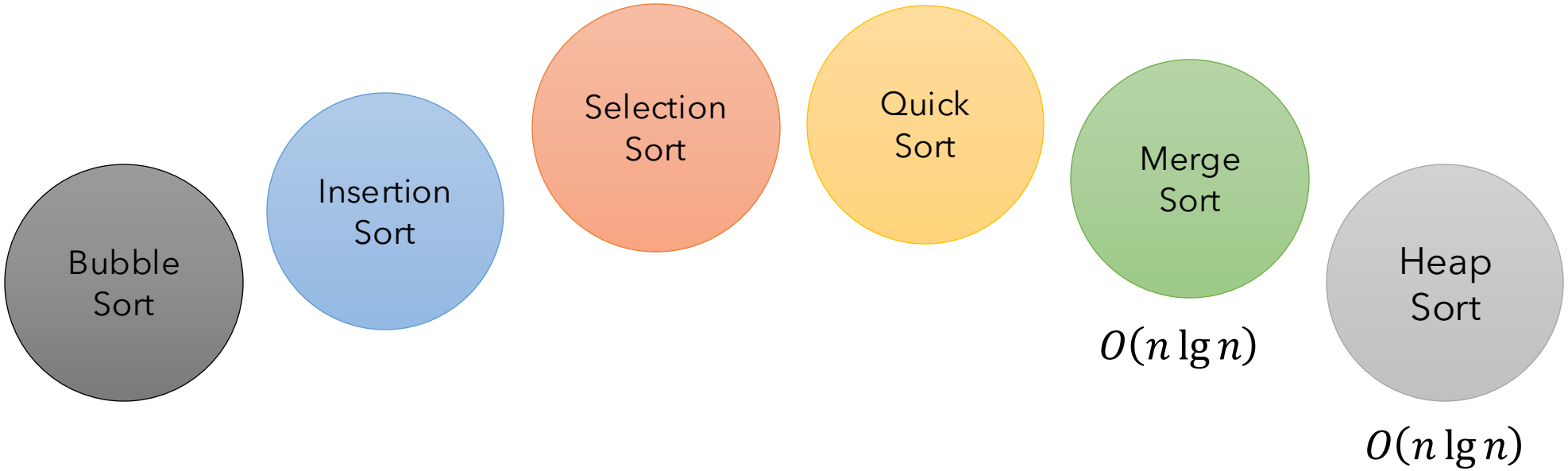
It combines the strengths of multiple algorithms, making it suitable for a wide range of input data.



Simplicity:

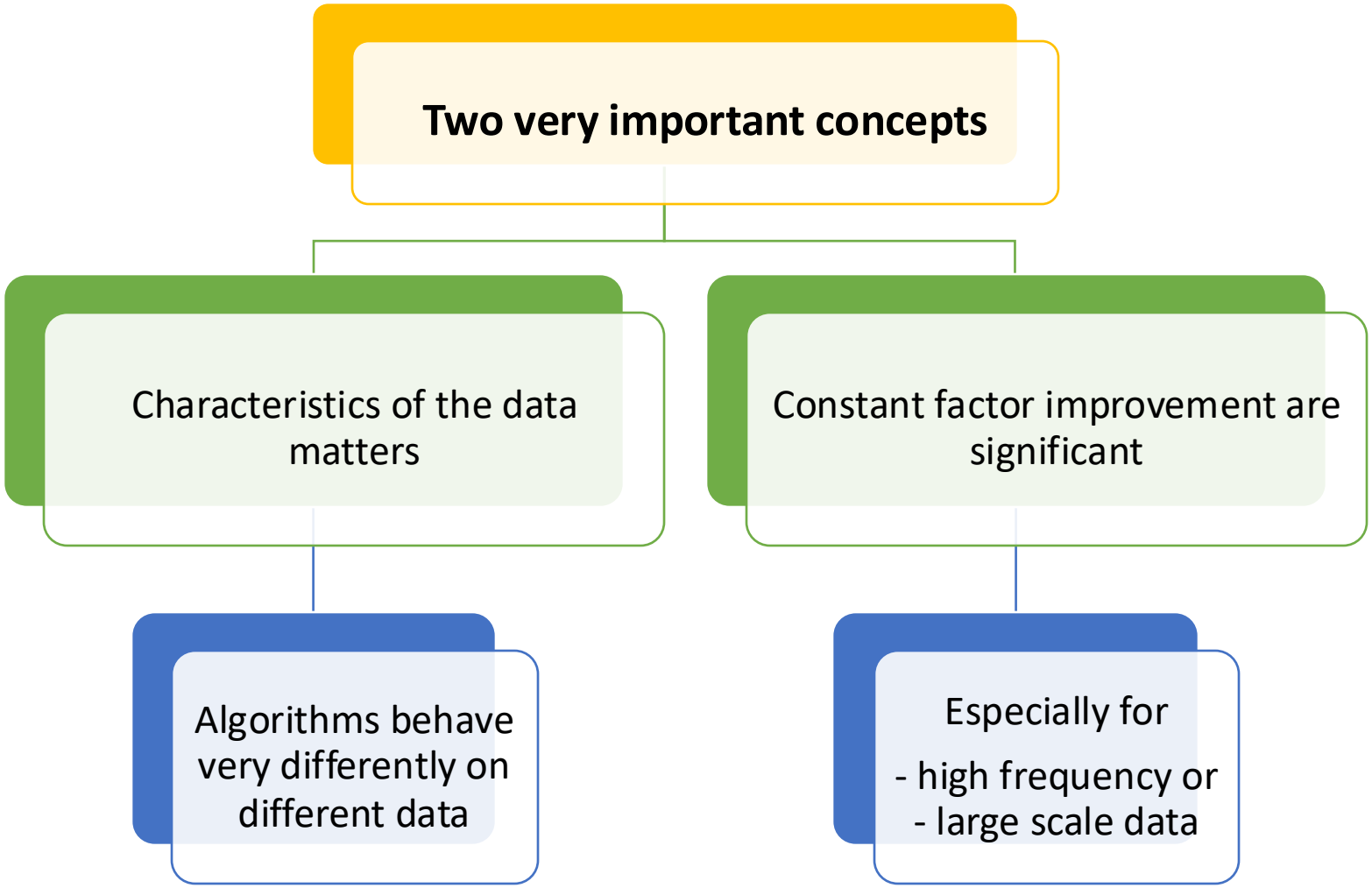
The algorithm's logic is relatively straightforward, making it easy to understand and implement.

Lower Bounds For Sorting



How faster can we sort?

Today


Two very important concepts
Characteristics of the data matters
Constant factor improvement are significant
Algorithms behave very differently on different data
Especially for
- high frequency or
- large scale data

Lower Bounds For Sorting

How fast can we sort?

- We will prove a lower bound, then beat it by playing a different game

Comparison sorting

- The only operation that may be used to gain order information about a sequence is comparison of pairs of elements (insertion sort, selection sort, merge sort, quicksort, heapsort)

Lower bounds

- $\Omega(n)$ to examine all the input.
- All sorts seen so far are $\Omega(n \lg n)$.
- We'll show that $\Omega(n \lg n)$ is a lower bound for comparison sorts.

Lower Bounds For Sorting (continued)

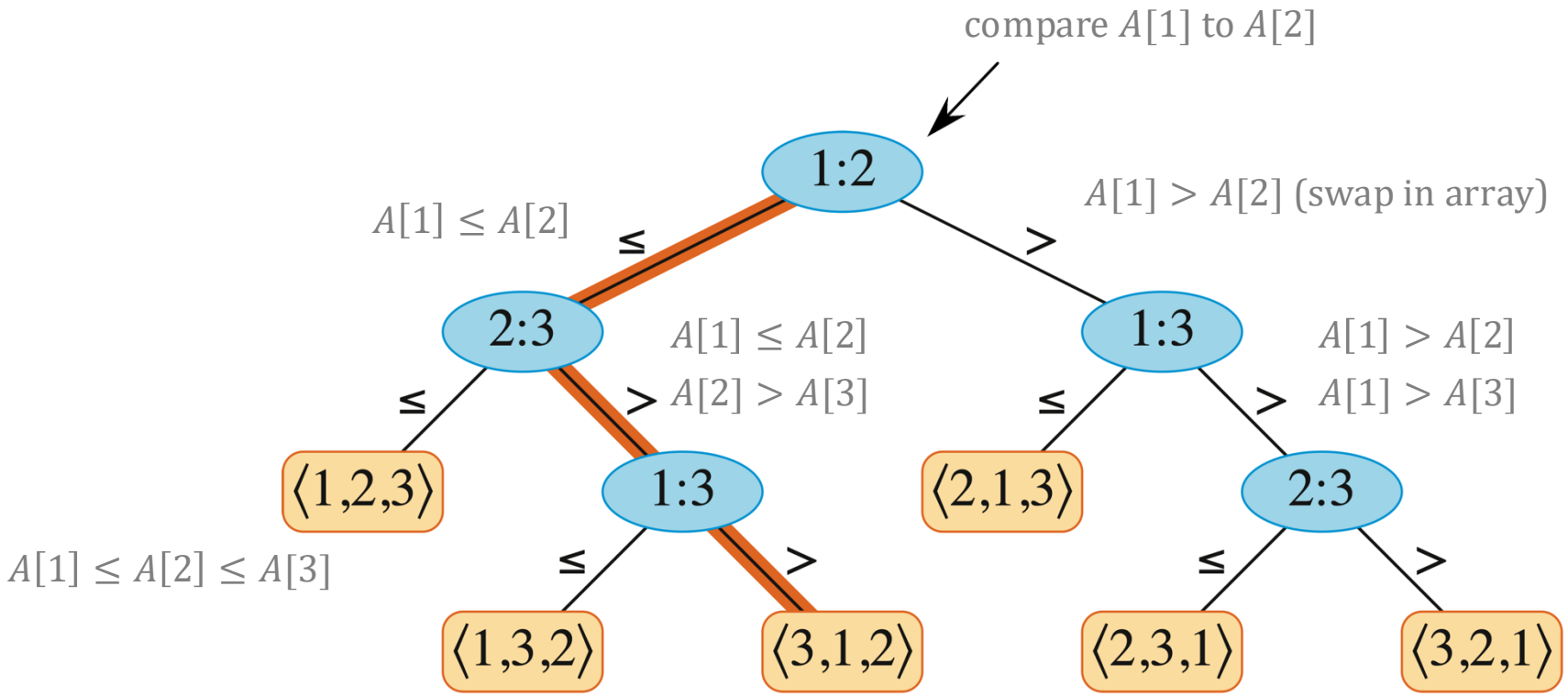
Decision tree

- Abstraction of any comparison sort.
- Represents comparisons made by
  - a specific sorting algorithm
  - on inputs of a given size.
- Abstracts away everything else: control and data movement.
- We're counting ***only*** comparisons.

Lower Bounds For Sorting (continued)

With no loss of generality, we assume all elements are distinct!

For insertion sort on 3 elements:



Note: This example is for ascending order.

Lower Bounds For Sorting (continued)

Now,
Let's find a *lower bound* for the *height of any decision tree that sorts n elements*

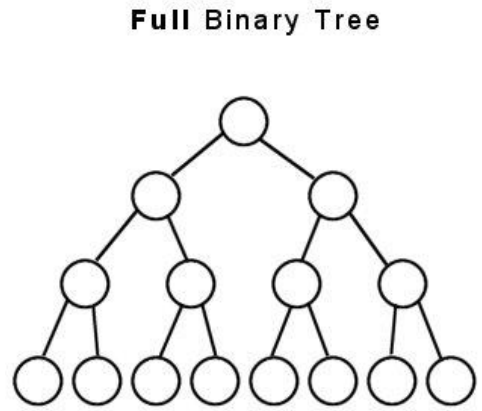
How many leaves on the decision tree?

Minimum:

- There are $\geq n!$ leaves, you ask why?
  - because every permutation appears at least once.

Maximum:

in a complete k-ary tree, there are k^h leaves



Lower Bounds For Sorting (continued)

- **Let's do this on whiteboard!**

Use Stirling's approximation: $n! > (n/e)^n$

Lower Bounds For Sorting (continued)

Proof Let the decision tree have height h and l reachable leaves.

- $l \geq n!$
- $l \leq 2^h$ (see Section B.5.3: in a complete k-ary tree, there are k^h leaves)
- $n! \leq l \leq 2^h$ or $2^h \geq n!$
- Take logarithms: $h \geq \lg(n!)$
- Use Stirling's approximation: $n! > (n/e)^n$ (by equation (3.23))

$$\begin{aligned} h &\geq \lg(n/e)^n \\ &= n \lg(n/e) \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) \ . \end{aligned}$$

Counting Sort

Counting sort

Depends on a *key assumption*: numbers to be sorted are integers in $\{0, 1, \ldots, k\}$.

Input: $A[1:n]$, where $A[j] \in \{0, 1, \ldots, k\}$ for $j = 1, 2, \ldots, n$. Array A and values n and k are given as parameters.

Output: $B[1:n]$, sorted.

Auxiliary storage: $C[0:k]$

Counting Sort - Pseudocode

```
COUNTING-SORT(A, n, k)
1   let B[1 : n] and C[0 : k] be new arrays
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to n
5       C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i − 1]
9   // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

Counting Sort (continued)

Do an example for $A = \langle 2_1, 5_1, 3_1, 0_1, 2_2, 3_2, 0_2, 3_3 \rangle$. *[Subscripts show original order of equal keys in order to demonstrate stability.]*

i	0	1	2	3	4	5
$C[i]$ after second **for** loop	2	0	2	3	0	1
$C[i]$ after third **for** loop	2	2	4	7	7	8

Sorted output is $\langle 0_1, 0_2, 2_1, 2_2, 3_1, 3_2, 3_3, 5_1 \rangle$.

Counting Sort (continued)

```
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

Counting Sort is STABLE

How this piece of code does the trick?

Counting Sort (continued)

```
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

Idea:

After the third **for** loop, $C[i]$ counts how many keys are less than or equal to i . If all elements are distinct, then an element with value i should go into $B[i]$.

But if elements are not distinct, by examining values in A in reverse order, the last **for** loop puts $A[j]$ into $B[C[A[j]]]$ and then decrements $C[A[j]]$ so that the next time it finds element with the same value as $A[j]$, that element goes into the position of B just before $A[j]$.

Sample Quiz

Suppose A = [4,2,1,5,1,4,1,2,1,5,1,4]. Determine C.

```
COUNTING-SORT(A, n, k)
1   let B[1:n] and C[0:k] be new arrays
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to n
5       C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i − 1]
9   // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

The correct answer is **C = [0,5,7,7,10,12]**.

2 Minutes

Counting Sort (continued)

Counting sort is ***stable*** (keys with same value appear in same order in output as they did in input) because of how the last loop works.

Analysis

Pick a Sorting Algorithm (Sample Quiz Question)

Let's consider two scenarios:

A. Sorting 10,000 students based on their letter grade (A, A-, B+, B, B-, C+, C)
B. Sorting 100 students based on the average calories they consume daily

Which sorting algorithm you'd use in each case?

2 Minutes

Counting Sort (continued)

Analysis

$\Theta(n + k)$, which is $\Theta(n)$ if $k = O(n)$.

2 Minutes

How long it takes for “Counting Sort” to handle the ”array”?

- $O(n)$
- $O(n \log n)$
- $O(n^2)$

```
def generate_input(n):
    array = []
    for i := 1 to n:
        array.append(i*i);
    shuffle(array)
    return array
```

Bucket Sort

Assumes that the input is generated by a random process that distributes elements uniformly and independently over [0, 1).

Idea

- Divide [0, 1) into n equal-sized *buckets*.
- Distribute the n input values into the buckets. *[Can implement the buckets with linked lists; see Section 10.2.]*
- Sort each bucket.
- Then go through buckets in order, listing elements in each one.

Bucket Sort (continued)

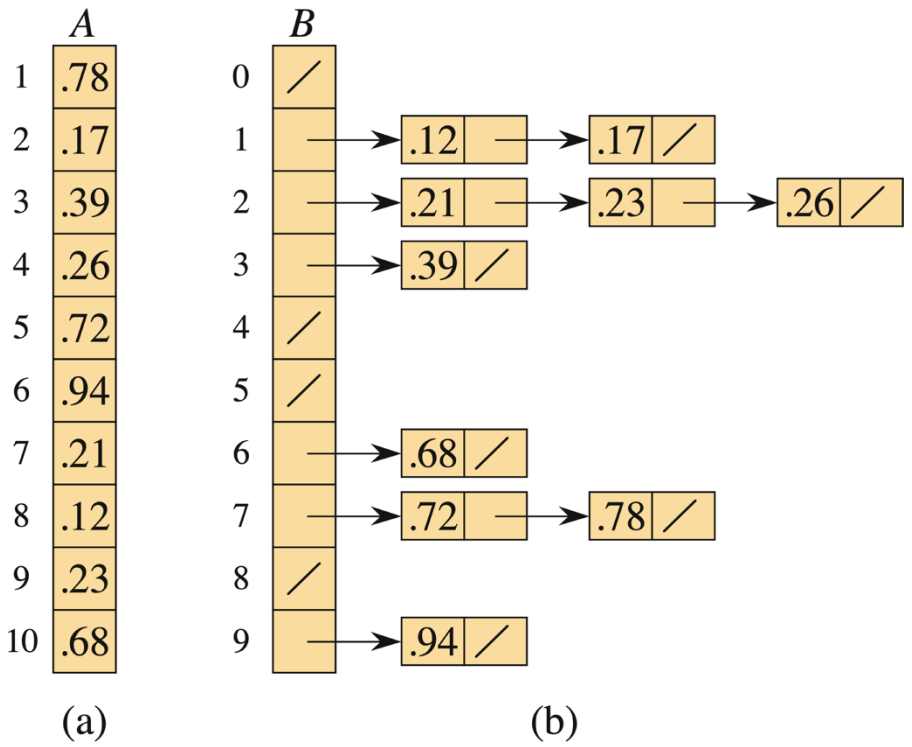
Input: $A[1:n]$, where $0 \le A[i] < 1$ for all i.

Auxiliary array: $B[0:n-1]$ of linked lists, each list initially empty.

BUCKET-SORT(A, n)

```
1  let B[0 : n − 1] be a new array
2  for i = 0 to n − 1
3        make B[i] an empty list
4  for i = 1 to n
5        insert A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7        sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
9  return the concatenated lists
```

Bucket Sort – Example (continued)



- The buckets are shown after each has been sorted.

Bucket Sort – Analysis

- Relies on no bucket getting too many values.
- All lines of algorithm except insertion sorting take $\Theta(n)$ altogether.
- Intuitively, if each bucket gets a constant number of elements, it takes $O(1)$ time to sort each bucket $\Rightarrow$ $O(n)$ sort time for all buckets.
- We "expect" each bucket to have few elements, since the average is 1 element per bucket.
- But we need to do a careful analysis.

Bucket Sort – Analysis (continued)

Define a random variable:

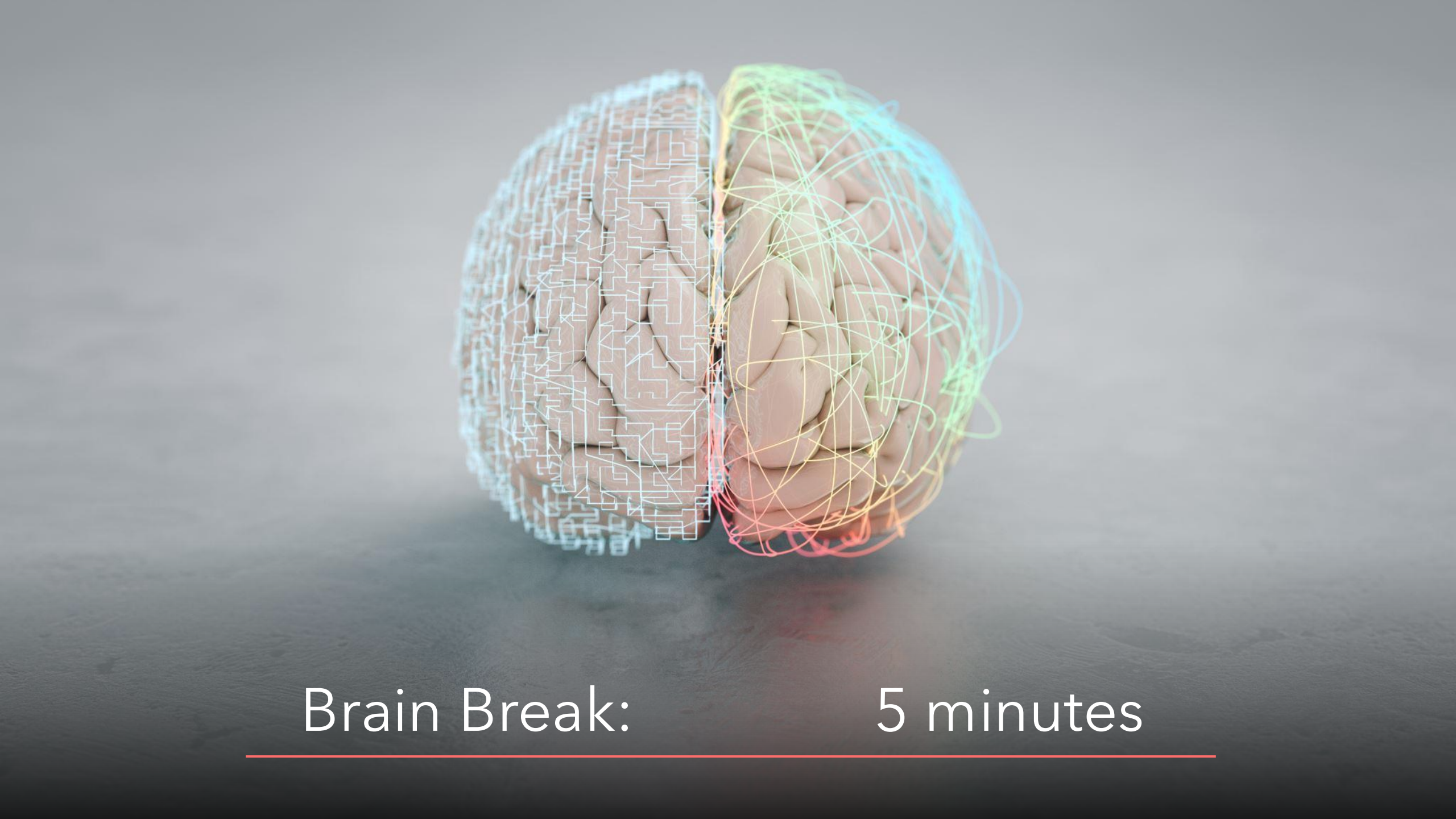
n_i = the number of elements placed in bucket $B[i]$.

Because insertion sort runs in quadratic time, bucket sort time is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

Take expectations of both sides:

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \qquad \text{(linearity of expectation)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}\left[n_i^2\right]) \qquad (\mathrm{E}[aX] = a\mathrm{E}[X]) \end{aligned}$$



Brain Break: 5 minutes

Claim

View each n_i as number of successes in n Bernoulli trials
Success occurs when an element goes into bucket $B[i]$.

- Probability p of success: $p = 1/n$.
- Probability q of failure: $q = 1 - 1/n$.

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] \\
&= \mathrm{E}\left[X^2 - 2X\mathrm{E}[X] + \mathrm{E}^2[X]\right] \\
&= \mathrm{E}[X^2] - 2\mathrm{E}[X\mathrm{E}[X]] + \mathrm{E}^2[X] \\
&= \mathrm{E}[X^2] - 2\mathrm{E}^2[X] + \mathrm{E}^2[X] \\
&= \mathrm{E}[X^2] - \mathrm{E}^2[X] \ .
\end{aligned}$$

$$\mathrm{E}[X^2] = \mathrm{Var}[X] + \mathrm{E}^2[X]$$

Claim

Binomial distribution counts number of successes in n trials:

$$\mathrm{E}[X_i] = p \cdot 1 + q \cdot 0 = p$$

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} p \\
&= np.
\end{aligned}$$

$$\mathrm{Var}[X_i] = \mathrm{E}[X_i^2] - \mathrm{E}^2[X_i]$$

$$\mathrm{Var}[X_i] = p - p^2 = p(1-p) = pq$$

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{Var}[X_i] \\
&= \sum_{i=1}^{n} pq \\
&= npq\ .
\end{aligned}$$

Claim

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \, .$$

Take expectations of both sides:

$$\begin{aligned}
\mathrm{E}\left[T(n)\right] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] && \text{(linearity of expectation)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}\left[n_i^2\right]) && (\mathrm{E}\left[aX\right] = a\mathrm{E}\left[X\right])
\end{aligned}$$

$$\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \mathrm{Var}\left[n_i\right] + \mathrm{E}^2\left[n_i\right] \\
&= (1 - 1/n) + 1^2 \\
&= 2 - 1/n
\end{aligned}$$

Therefore:

$$\begin{aligned}
\mathrm{E}\left[T(n)\right] &= \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) \\
&= \Theta(n) + O(n) \\
&= \Theta(n)
\end{aligned}$$

Week 4 Quiz – Q3

If N=3, there are $3^3 = 27$ ways the three objects can be placed into the three buckets.

- The probability of this bucket **contains exactly 0 objects**:
  - $8/27 = 2/3 \times 2/3 \times 2/3$
- The probability of this bucket **contains exactly 1 objects**:
  - $12/27 = (1/3 \times 2/3 \times 2/3) + (2/3 \times 1/3 \times 2/3) + (2/3 \times 2/3 \times 1/3)$
- The probability of this bucket **contains exactly 2 objects**:
  - $6/27 = (1/3 \times 1/3 \times 2/3) + (1/3 \times 2/3 \times 1/3) + (2/3 \times 1/3 \times 1/3)$
- The probability of this bucket **contains exactly 3 objects**:
  - $1/27 = (1/3 \times 1/3 \times 1/3)$

Week 4 Quiz – Q3

What is the probability that a bucket contains exactly one object?

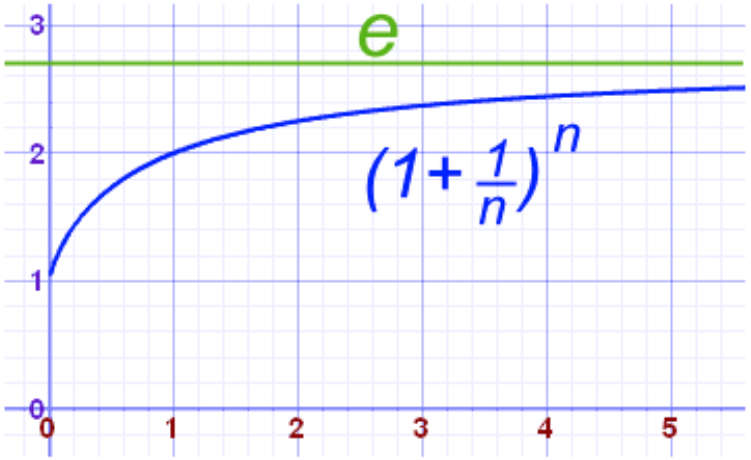
This probability is $N \times (1/N) \times ((N-1)/N)^{N-1}$

Which is equal to:

$= 1 / (1 + 1/(N-1))^{N-1}$

$= 1/e$

For example, the value of $(1 + 1/n)^n$ approaches ***e*** as n gets bigger and bigger:



n	$(1 + 1/n)^n$
1	2.00000
2	2.25000
5	2.48832
10	2.59374
100	2.70481
1,000	2.71692
10,000	2.71815
100,000	2.71827

https://www.mathsisfun.com/numbers/e-eulers-number.html

Week 4 Quiz – Q3

Binomial Distribution with parameters n and p is the discrete probability distribution of the number of successes in a sequence of n independent experiments, each asking a yes–no question

Let's Toss a Coin!

Toss a fair coin **three times** ... what is the chance of getting exactly **two Heads**?

Using **H** for heads and **T** for Tails we may get any of these 8 **outcomes** :

Week 4 Quiz – Q3

- $p_n(k)$ denote the probability that a bucket contains <u>exactly</u> k out of the n objects.

Probability Mass Function for a Binomial

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

Probability that our variable takes on the value k

For sufficiently large n, we have:
n! / (k! * (n-k)!) = 1/k!

For sufficiently large n:

$p_n(k) = 1/k! \times 1/e$

So what does this mean?

$p_n(0) + p_n(1) + p_n(2) = 1/e + 1/e + 1/2e = 0.9196$, so **about 92%** of buckets contain at most 2 elements.

$p_n(0) + p_n(1) + p_n(2) + p_n(3) + p_n(4) = 1/e + 1/e + 1/2e + 1/6e + 1/24e = 0.9963$, so about 99.6% of buckets contain at most 4 elements.

Week 4 Quiz – Q3

- $p_n(k)$ denote the probability that a bucket contains <u>exactly</u> k out of the n objects.

Proof. This is a refinement of the argument above. Note that the probability that an item is in a given bucket is $p = 1/b$ and the probability it is not in that bucket is $q = 1 - p = (b-1)/b$. If we have a particular choice of k items, the probability that exactly these items are in a given bucket is $p^k q^{n-k}$, because those k items are each in the bucket, accounting for the p^k factor, and the remaining $n-k$ items are *not* in the bucket, accounting for the q^{n-k} factor. There are $\binom{n}{k}$ independent choices of exactly k items to put in the bucket, so the probability that the given bucket has size k is the sum of $p^k q^{n-k}$ over all these independent ways of selecting k items:

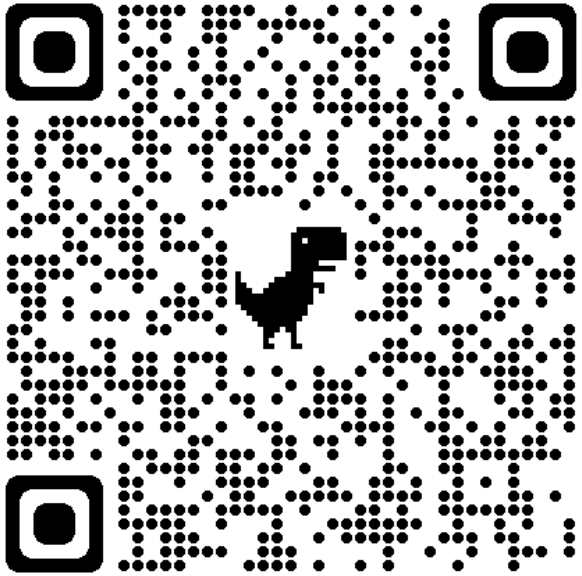
Probability Mass Function for a Binomial

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

Probability that our variable takes on the value k

OLD Quiz (Demo)

- https://colab.research.google.com/drive/19l7cI2jVaqGXSbzePoYQYwCZ4tXY9U7U?usp=sharing
- https://chrispiech.github.io/probabilityForComputerScientists/en/part2/binomial/



10 Minutes

Overview

The ***selection problem***:

Input: A set A of n distinct numbers and a number i, with $1 \le i \le n$.

Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements in A.
In other words, the ith smallest element of A.

Easy to solve the selection problem in $O(n \lg n)$ time:

- Sort the numbers using an $O(n \lg n)$-time algorithm, such as heapsort or merge sort.
- Then return the ith element in the sorted array.

- **How do we do this?**
  - **Brute Force?**

We can solve the selection problem in O(n lg n) time simply by sorting the numbers using heapsort or merge sort and then outputting the i^{th} element in the sorted array.

Minimum & Maximum

- We can easily obtain an upper bound of $n-1$ comparisons for finding the minimum of a set of n elements.
- Examine each element in turn and keep track of the smallest one.
- This is the best we can do, because each element, except the minimum, must be compared to a smaller element at least once.
- The following pseudocode finds the minimum element in array $A[1:n]$:

```
MINIMUM(A, n)
1  min = A[1]
2  for i = 2 to n
3      if min > A[i]
4          min = A[i]
5  return min
```

- **Can we do better?**

Minimum & Maximum

- Some applications need both the minimum and maximum of a set of elements.
- For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display. To do so, the program must first find the minimum and maximum of each coordinate.
- A simple algorithm to find the minimum and maximum is to find each one independently. There will be $n - 1$ comparisons for the minimum and $n - 1$ comparisons for the maximum, for a total of $\mathbf{2n - 2}$ **comparisons**. This will result in $\Theta(n)$ time.

Simultaneous Minimum & Maximum (continued)

- In fact, at most $3\lfloor n/2 \rfloor$ comparisons suffice to find both the minimum and maximum:
- Maintain the minimum and maximum of elements seen so far.
- Don't compare each element to the minimum and maximum separately.
- Process elements in pairs.
- Compare the elements of a pair to each other.
- Then compare the larger element to the maximum so far, and compare the smaller element to the minimum so far.
- This leads to only **3 comparisons for every 2 elements**.

Class Exercise

Problem Statement:

- Given a set of points in the 2D plane, your task is to find the smallest axis-aligned rectangle that can contain all the points. The rectangle's sides must be parallel to the axes. Return the area of this rectangle.

Input: points = [[1,2], [2,3], [3,4]] **Output:** 2

Explanation: The smallest rectangle that can contain all points is formed by [1,2] and [3,4], giving an area of 2.

Constraints:

- 1 <= points.length <= 10^5
- points[i].length == 2
- -10^4 <= points[i][0], points[i][1] <= 10^4

Note: You may assume that no two points are the same.

Link is also available on Canvas:
Modules: Week 3 - Class Prep Reading

🧠 Class Activity - Labs

Class Exercise

939 Minimum Area Rectangle

Simultaneous Minimum & Maximum (continued)

- Setting up the initial values for the min and max depends on whether n is odd or even.
- If n is even, compare the first two elements and assign the larger to max and the smaller to min. Then process the rest of the elements in pairs.
- If n is odd, set both min and max to the first element. Then process the rest of the elements in pairs.

Analysis Of The Total Number Of Comparisons

- If n is even, do 1 initial comparison and then $3(n-2)/2$ more comparisons.

$$\begin{aligned}
\#\text{ of comparisons} &= \frac{3(n-2)}{2} + 1 \\
&= \frac{3n-6}{2} + 1 \\
&= \frac{3n}{2} - 3 + 1 \\
&= \frac{3n}{2} - 2\,.
\end{aligned}$$

- If n is odd, do $3(n-1)/2 = 3\lfloor n/2 \rfloor$ comparisons.

In either case, the maximum number of comparisons is $\leq 3\lfloor n/2 \rfloor$.

Some LeetCode problems where this approach can be beneficial:

LeetCode #462: Average Salary Excluding the Minimum and Maximum Salary

LeetCode #414: Third Maximum Number

LeetCode #1299: Replace Elements with Greatest Element on Right Side

LeetCode #485: Max Consecutive Ones

LeetCode #1109: Three Consecutive Odds

LeetCode #1133: Check if Array Is Sorted and Rotated

LeetCode #189: Rotate Array

Next Week: Dynamic Programming

📚 Required Prep:

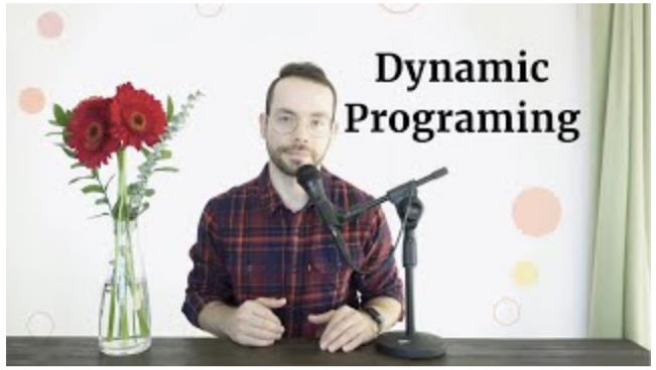
Option 1:

Read Chapter 14 of the Course Textbook (4th Edition) - Dynamic Programming

Make sure you do a close and careful reading of Section 14.1, Section 14.3, and Section 14.4

You are welcome to skim/skip the other two sections: Section 15.2, Section 15.5

Option 2:



Just watch: https://www.youtube.com/watch?v=5zpZmORdi0Y

Next Week
Your 1st Quiz

Covering topics of Weeks 1-3

e.g., Algo Complexity, DAC, Master Theorem, Sorting, etc

Best strategy: Start review our class discussions, sample quiz questions in each lectures, and finally your HW

Few MCQ + 1 Short-Medium Answer

It will have a time limit of 30 ∓ 10 minutes

Feel free to bring a Cheetsheet:

A4, single-sided

Handwritten

Note: Your Cheetsheet won't be allowed if it fails to meet the above conditions.

Homework 2 Preview

Questions?

Northeastern University