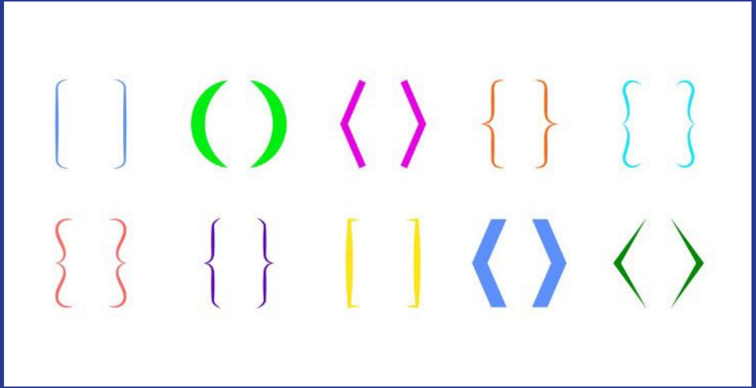
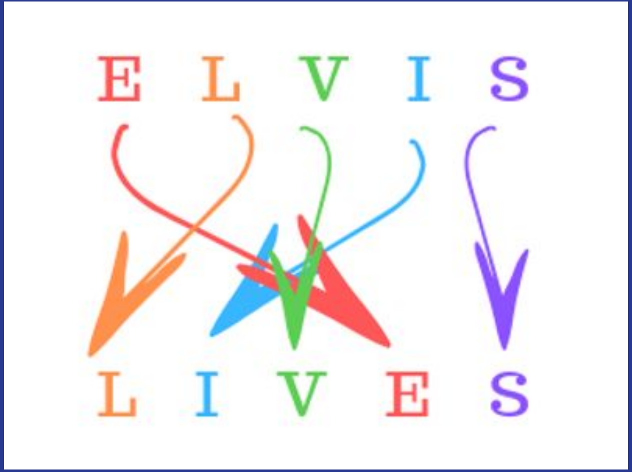
CS 9

Week 2 Problems

Andrew Benson
Ian Tullis





3 Problems

- Problem 2-1 has a video walkthrough on Canvas, generously created by Ian
- Problem 2-2 will be discussed in-class
- Problem 3-3 will have a written explanation at the end of the slides

- Pick a problem to work on for ~35 minutes. If possible, try to write code to implement your solution. I generally aim to spend about half the time brainstorming solutions and half the time writing code.

⭐ Problem 2-1: Balanced Parentheses

- You are given a string where each character is one of:
  `( ) [ ] { } < >`
- Your goal is to determine whether the string is "balanced" - that is, each open bracket `(` or `[` or `{` or `<` is correctly closed in the right order by its corresponding closing bracket `)` or `]` or `}` or `>`. Hopefully this matches your intuitive sense of what feels balanced and valid.
  - Part 1: Solve this for the case where only `( )` are allowed
  - Part 2: Extend your solution for all given bracket types
- Input is the string, output is `true` or `false`
- Examples:
  - Input: "`((()()))`" Output: `true`
  - Input: "`)(`" Output: `false`
  - Input: "`()[]<{}>`" Output: `true`
  - Input: "`(<)>`" Output: `false`

⭐ Problem 2-1: Balanced Parentheses

- Some thoughts (only if you need a hint!):
  - Do some examples. The given specifications for balanced parentheses aren’t very rigorous, so try to refine how you intuitively can determine whether a string is balanced or not.
  - What must be true about the first character?
  - Given an open bracket, how do you know which is its corresponding closing bracket?
  - For Part 2: if you’re having trouble keeping track of…something…maybe there’s a data structure that suits your needs?

⭐⭐ Problem 2-2: Separating Anagrams

- You are given a list of lowercase English strings (both count and length could be large, like 10^4). It's possible that some of these strings are anagrams of each other. Our goal is to divide up the strings into groups where within each group, the strings are anagrams of each other.
- Input: A collection of strings, output: A collection of collections of strings, where within each inner collection, each string is an anagram of every other string in the collection
- Examples:
  - Input: `{"rat", "chain", "tar", "soon", "china", "art"}`
    Output: `{{"art", "rat", "tar"}, {"soon"}, {"china", "chain"}}`
  - Input: `{"bone", "bones"}`
    Output: `{{"bones"}, {"bone"}}`

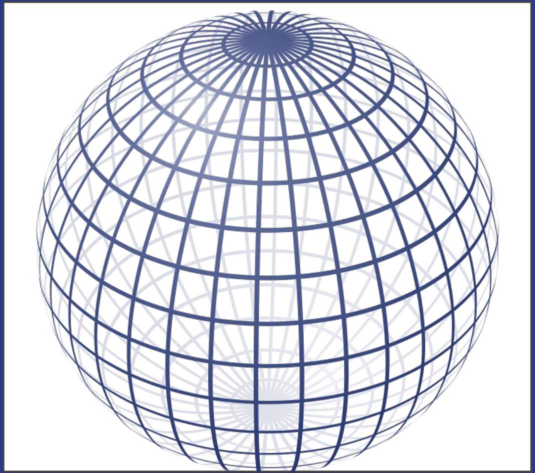
⭐⭐ Problem 2-2: Separating Anagrams

- Some thoughts (only if you need a hint!):
    - What's a brute force way to do this?
    - How can you determine whether two words are anagrams?
    - Note that the strings are lowercase English - that limits the possible characters that appear.
    - What if you had `n` words that are supposedly all anagrams of each other - how could you verify that quickly?
    - If you have already grouped the first `k` words into anagram groups, and you're looking at the `(k+1)`th, could you determine if it's part of a previously analyzed anagram group?

⭐ ⭐ ⭐ Problem 2-3: Sphere*

- Part 1: How would you write code to take a *uniformly random* sample from the *outer shell* of a sphere of radius 1? Can you find a randomized method that works?

- Part 2: Does your method generalize to, e.g., 100-dimensional "spheres"? Explain why or why not.



**anagramming this word is not recommended*

The following slides discuss solutions

Separating Anagrams

- How do we determine if strings are anagrams?

Separating Anagrams

- How do we determine if strings are anagrams?
- Anagrams are strings where we don’t care about order, so they’re just bags of letters - two strings are anagrams if they are the same bag of letters

Separating Anagrams

- How do we determine if strings are anagrams?
- Anagrams are strings where we don't care about order, so they're just bags of letters - two strings are anagrams if they are the same bag of letters
    - Idea 1: Impose a standard order by sorting the strings - two strings are anagrams iff their sorted versions are equal

$$\text{state} \longrightarrow \text{aestt}$$

$$\text{taste} \longrightarrow \text{aestt}$$

Separating Anagrams

- How do we determine if strings are anagrams?
- Anagrams are strings where we don't care about order, so they're just bags of letters - two strings are anagrams if they are the same bag of letters
    - Idea 2: Compare the bags of letters directly - bags of letters are frequency count dictionaries (hash tables mapping letters to the number of appearances)

$$\text{state} \longrightarrow \{a: 1, e: 1, s: 1, t: 2\}$$

$$\text{taste} \longrightarrow \{a: 1, e: 1, s: 1, t: 2\}$$

Separating Anagrams

- What would a brute force solution be?

Separating Anagrams

- What would a brute force solution be?
  - Keep track of all your anagram collections, perhaps in a list
  - For each string, see if it matches of these collections by seeing if it’s an anagram of one of its strings
- What’s the complexity?

Separating Anagrams

- What would a brute force solution be?
  - Keep track of all your anagram collections, perhaps in a list
  - For each string, see if it matches of these collections by seeing if it's an anagram of one of its strings
- What's the complexity?
  - Determining if anagrams: O(k) where k is the length of the longest string
  - In the worst case, there are O(n) anagram collections (where n = number of strings), so $O(n^2k)$

Separating Anagrams

- Where’s the redundancy / inefficiency in this brute force solution?

Separating Anagrams

- Where’s the redundancy / inefficiency in this brute force solution?
    - We have to “linear search” through all collections to “lookup” what collection a string is part of
        - Can we do this faster?

Separating Anagrams

- Where’s the redundancy / inefficiency in this brute force solution?
    - We have to “linear search” through all collections to “lookup” what collection a string is part of
        - Can we do this faster?
            - Hash tables provide O(1) lookup. Given a string, can we somehow extract “a key” that allows us to lookup the anagram collection?

Separating Anagrams

- Where’s the redundancy / inefficiency in this brute force solution?
    - We have to “linear search” through all collections to “lookup” what collection a string is part of
        - Can we do this faster?
            - Hash tables provide O(1) lookup. Given a string, can we somehow extract “a key” that allows us to lookup the anagram collection?
                - Yes. We could use the sorted string, or the letter frequency count dictionary. (The latter needs to be made hashable.)

Separating Anagrams

- Sorted string as key approach

$$\text{aestt} \longrightarrow \{\text{state, taste}\}$$

$$\text{eilsv} \longrightarrow \{\text{lives, elvis}\}$$

- Given a new string, we’d sort it, and lookup in this map whether we have an anagram for that anagram collection
- Complexity: O(nk logk)

Separating Anagrams

- Letter frequency dictionary as key approach
  - Problem: in most languages, dictionaries are not hashable, so they can’t be keys of a map
  - Solution: Serialize it into a string somehow! (Why is this constant time?)

a1e1s1t2 $\longrightarrow$ {state, taste}

e1i1l1s1v1 $\longrightarrow$ {lives, elvis}

- Complexity: O(nk)

A possible implementation

```
def serialize_to_lettercount(s):
    freq = {}
    for letter in s:
        old_count = freq.get(letter, 0)
        freq[letter] = old_count + 1
    res = ""
    for letter in sorted(freq.keys()):
        res += letter
        res += str(freq[letter])
    return res
```

```
def separate_anagrams(strings):
    anagrams = {}
    for s in strings:
        key = serialize_to_lettercount(s)
        anagram_collection = anagrams.get(key, set())
        anagram_collection.add(s)
        anagrams[key] = anagram_collection
    return list(anagrams.values())
```

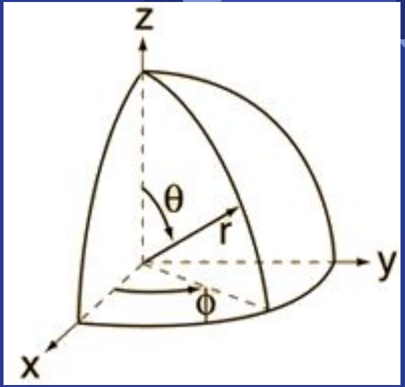
Don't let us forget to say the password

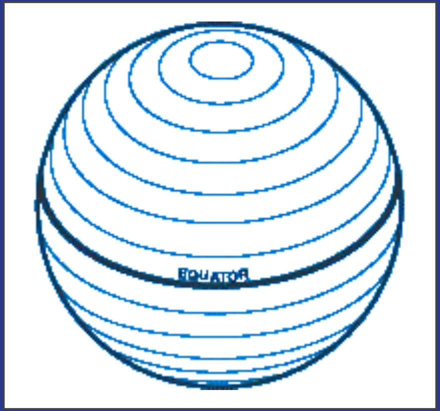
Sphere: solution

It may seem that we can choose two polar coordinates θ and Φ, each uniformly at random from [0, 2π], and then take the point in that direction that is a distance of 1 away from the origin.

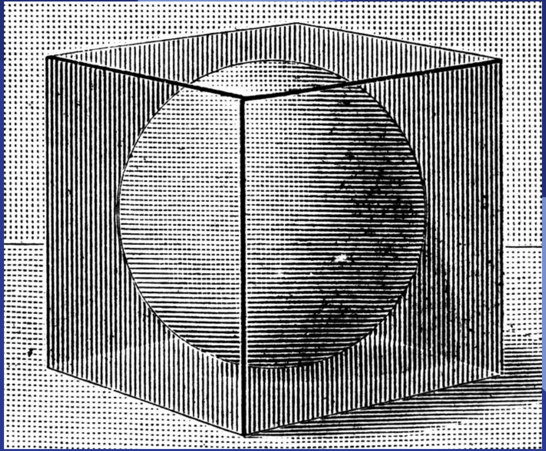
But this is not a uniform sample! Imagine choosing a latitude on the Earth's surface, for instance, then choosing a point on that latitude. It's clearly wrong for the equator and the North Pole to get the same weight!

We can maybe try to correct for this by setting up the right multivariable integral, but, ew, calculus.





What if… we imagine the sphere being inscribed in a 2 x 2 x 2 cube, and then choose a point uniformly at random from within the *cube*? (which is easy – just take uniform random x, y, z in the interval [0, 2]).



We check if our point is *in* the sphere. Also easy – just see if $x^2 + y^2 + z^2 \leq 1$. If the point is outside the sphere, we just try again until we get one inside the sphere.

Then we draw a ray from the center of the sphere through our point and continue until we hit the shell (forming a radius), and take that new point as our sample.

This works because we are sampling the volume within the sphere uniformly, and there is a unique radius from each point back to the origin. By symmetry, we are no more likely to land on one radius versus another.

For a 3D sphere, the chance of getting a point within the sphere is the volume of a unit sphere divided by the volume of a 2x2x2 cube, i.e., $(4\pi(1^3)/3) / 2^3 \approx 4.19 / 8$. So we have a little over a 50% chance of getting a point in the sphere, and we can just try until we get one.

But in higher dimensions, the hypersphere occupies an increasingly small fraction of the hypercube! E.g., a 10-dimensional unit sphere takes up only about 0.25% of the area of the cube. As the dimension increases further, this gets *much* worse, until we have a negligible chance of sampling a point in the sphere.

This situation comes up in machine learning! It's an example of the so-called "curse of dimensionality". CS168 has more detail…

Even for the 3D sphere, another issue (in practice) is that a computer can't really choose an *arbitrary* point within the cube. Only certain discrete points are possible, depending on what level of precision the computer can handle. Because these allowable points are in a lattice-like arrangement, certain radii are in fact much more likely to be hit than others…