1 Python Cost Model

1.1 Python Lists

In Python, lists are implemented as arrays. This means that they are stored as a contiguous block of memory. With this is mind, let's analyze the runtimes of some of the basic operations of Python lists.

1. APPEND: This just involves placing the element in the next block of memory after the array, so it takes $O(1)$ time. For now, we will not worry about how Python ensures that the necessary block of memory is available.
2. SEARCH: In the worst case, this involves scanning the entire list, so it takes $O(n)$.
3. REMOVE: When we remove the element, we have to move all of the following elements over so that the array does not contain empty spaces, so this takes $O(n)$ in the worst case.

1.2 Python Dictionaries

In Python, dictionaries are implemented using hash tables. We will discuss hash tables later in the class, so for now you are not responsible for understanding how dictionaries are able to support these operations in these runtimes.

1. APPEND: $O(1)$
2. SEARCH: $O(1)$
3. REMOVE: $O(1)$

1.3 Comparison

Given that the runtimes of the dictionary operations are strictly better than those of a list, why would we ever want to use a list? Well, unlike dictionaries, lists are ordered. So, when we are in a situation where the order of the elements matters, we probably want to use a list.

1.4 Tips and Tricks

- Arithmetic operations (addition, multiplication, etc.) scale linearly in the number of digits (i.e. the logarithm) of numbers. For now, these operations can be considered constant.
- Taking slices of a list is linear in the length of the list, so if youre recursing inside of a pre-existing list, just keep track of the range of indices instead ($O(n)$ vs $O(1)$).
- If you dont need to maintain an existing list, then for two lists, `L` and `M`, do `L.extend(M)` instead of `L+M`. The first takes $O(M)$ time while the latter takes $O(L + M)$ time.

- Dont delete from the beginning of a list. Every element has to be shifted, which takes $O(n)$ time. Consider using `collections.deque` if deleting from both ends is necessary.

2 Document Distance

2.1 Definition Review

Define D_i to be the vector of frequencies of words in document i. For now, we will represent D_i as a list of lists. Consider the following two documents

Document 1: "the dog"

Document 2: "the cat"

This will result in the following frequency vectors:

```
D_1 = [["the", 1], ["dog", 1]]

D_2 = [["the", 1], ["cat", 1]]
```

Next, we define the notion of an *inner product* of two frequency vectors. The inner product is the sum of the product of frequencies. So, the inner product of D_1 and D_2, denoted $D_1 \cdot D_2$, is given by

$$D_1 \cdot D_2 = 1 \cdot 1 + 1 \cdot 0 + 0 \cdot 1 = 1.$$

We also need to define the *norm* of of a frequency vector, denoted $|D_1|$, as the square root of the inner product of a frequency vector with itself. Specifically, we have

$$|D_1| = \sqrt{D_1 \cdot D_1}.$$

With these definitions in mind, we can now define the distance d between two documents as

$$d = \arccos\left(\frac{D_1 \cdot D_2}{|D_1||D_2|}\right).$$

2.2 Implementations

Next, we will analyze the runtimes of some implementations of computing the document distance. Say that we begin with two documents, represented as strings. We will use the following steps to compute the document distance:

1. Convert document string to frequency vector. This will be done using the method `get_word_counts(s)`
2. Then, we will need to compute some inner products. This will be done using the method `get_inner_product(D1, D2)`.
3. Finally, we need to plug these values into the document distance formula to get our final answer.

We will analyze implementations of `get_word_counts(s)` and `get_inner_product(D1, D2)` and state their runtimes.

2.2.1 Take 1: `get_word_counts`

```
def get_word_counts(s):
  words = s.split(" ")
  counts = []
  for word in words:
    found = False
    for i in range(len(counts)):
      if word == counts[i][0]:
        counts[i][1] += 1
        found = True
    if not found:
      counts.append([word, 1])
  return counts
```

Runtime: The outer for loop iterates through each word, and the inner for loop iterates through a list that can grow to be the same size as the number of words. Thus, we have that the runtime of the method is $O((\text{\# words})^2)$.

2.2.2 Take 1: `get_inner_product`

```
def get_inner_product(D1, D2):
  inner_product = 0
  for w1 in D1:
    for w2 in D2:
      if w1[0] == w2[0]:
        inner_product += w1[1] * w2[1]
  return inner_product
```

Runtime: The outer loop iterates through each element of the first frequency vector, and the inner loop iterates through each element of the second frequency vector. Thus, we have a runtime of $O(D_1 \cdot D_2)$. Note that for convenience, we are using D_1 to represent the number of unique words in the frequency vector D_1.

2.2.3 Take 1.5: `get_inner_product`

Instead of iterating through both frequency vectors, we could instead sort both D_1 and D_2. Then, we would only need to scan each list once. In this new algorithm, the runtime is dominated by the time to sort both of the lists. So, we have a runtime of $O(D_1 \log D_1 + D_2 \log D_2)$ (we have not yet discussed the runtime of sorting, so for now you can just take this runtime for granted).

Now, we think about a smarter way to store the frequency vectors. Instead of storing them as a list of lists, let's store them as a dictionary. So now, we would have

```
    D_1 = {"the": 1, "dog": 1}
    D_2 = {"the": 1, "cat": 1}
```

Let's write new implementations of our methods using our new frequency vectors.

2.2.4 Take 2: `get_word_counts`

```
def get_word_counts(s):
  words = s.split(" ")
  counts = {}
  for word in words:
    if word in counts:
      counts[word] += 1
    else:
      counts[word] = 1
  return counts
```

Runtime: Now, we only have to iterate through the list of words once, so our new runtimes is O(# words), much faster than before!

2.2.5 Take 2: `get_inner_product`

```
def get_inner_product(D1, D2):
  inner_product = 0
  for w in D1:
    if w in D2:
      inner_product += D1[w] * D2[w]
  return inner_product
```

Runtime: Now, we only have to iterate through the first frequency vector (because we can perform look-ups in the other frequency vector in constant time). So now, our runtime is given by $O(D_1)$, again, a significant improvement.