CS 120: Intro to Computer Programming II

In-Class Activity - 12 Complexity - Day 3

Activity 1 - Turn in this one

The following function is one of the components of a famous sorting algorithm, known as Merge Sort. Make an argument about how long it takes to run (hint: how much work does it take to fill each element of the output array). Then, verify that your analysis is correct by running the function with various inputs. Does it matter if the input arrays are the same size?

```
def merge(in1, in2, output):
    """merges two input arrays into an output array

       in1,in2 - input arrays, we assume both are sorted
       output  - output array, contents are junk
    """
    assert len(output) == len(in1)+len(in2)

    pos1 = 0
    pos2 = 0
    while pos1<len(in1) and pos2<len(in2):
        if in1[pos1] <= in2[pos2]:
            output[pos1+pos2] = in1[pos1]
            pos1 += 1
        else:
            output[pos1+pos2] = in2[pos2]
            pos2 += 1

    while pos1<len(in1):
        output[pos1+pos2] = in1[pos1]
        pos1 += 1

    while pos2<len(in2):
        output[pos1+pos2] = in2[pos2]
        pos2 += 1

    assert pos1+pos2 == len(output)
```

Solution: This function takes $O(n)$ time, where n is the number of elements in the **output**. This is because it takes constant-time work to find **each element to be copied.** Thus, overall it takes $O(n)$.

It doesn't matter whether one of the arrays is short; if that happens, we simply (might) spend a long time in the 2nd or 3rd `while` loops - but the argument holds for those loops, just as much as it does for the 1st.

(activity continues on next page)

Activity 2 - Turn in this one

Quicksort is another famous sorting algorithm. Its first step, `partition()`, is simple in concept but very tricky in practice! So let's look at a simplified version of it first.

Write a function which, given an input array and a "pivot" value, breaks the input array into two groups: values less than or equal to the pivot, and values greater than it. Build two arrays, and return them both, like this:

```
return arr1,arr2
```

Once you've written it, **make an argument** about its big-Oh cost, and then experimentally verify that you are correct. Does it matter how much data goes into `arr1` or `arr2` ?

Solution:

```
def not_really_partition_from_quicksort(data, pivot):
    arr1 = []
    arr2 = []

    for v in data:
        if v <= pivot:
            arr1.append(v)
        else:
            arr2.append(v)

    return arr1,arr2
```

This is $O(n)$ for the same reason as the previous activity: we do constant-time work for each value in the **input** array.

(activity continues on next page)

Activity 3 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

What makes Quicksort so cool - but tricky - is that `partition()` runs **without allocating any new memory!** Here's the actual partition function:

```
def partition(data):
    assert len(data) > 2

    # we will choose the middle element as the pivot.  A *real*
    # implementation of pivot would be trickier than this - but this
    # is complex enough for now.  The middle element works well if
    # the input data happens to already be sorted
    mid = len(data) // 2

    # move the pivot to the front
    data[0],data[mid] = data[mid],data[0]
    pivot = data[0]

    # how much have we sorted into the "small" and "large" arrays?
    i = 0
    j = len(data)

    # loop until the "small" and "large" arrays touch in the middle
    while i+1 < j:
        # advance i as much as possible
        while i+1 < j and data[i+1] <= pivot:
            i += 1

        # advance j as much as possible
        while j-1 > i and data[j-1] > pivot:
            j -= 1

        # if i and j aren't touching, then we know that i has been
        # stopped by a too-large value, and j has been stopped by
        # a too-small one.  Swap those two, and then both counters
        # can advance one step closer.
        if i+1 < j:
            assert i+1 < j-1
            data[i+1],data[j-1] = data[j-1],data[i+1]
            i += 1
            j -= 1

    # i,j now touch.  Move the pivot up into position, between them.
    data[0],data[i] = data[i],data[0]

    # return the index of the pivot position.
    return i
```

(instructions on next page)

Don't analyze the time performance of the function above (yet). Instead, **add some debug code** to the function above. Every time that `i` or `j` changes, print out the array in a funny way, like this (you'll need to write a function to do it):

```
val (val val val) val val val val (val)
```

Use the parentheses to keep track of how `i,j` change: the first set of parentheses should keep track of how far the "small" array has gotten, and the second set should keep track of the "large."

Then, run this partition algorithm on a moderate-size array (say, 20 or 30 elements) - something small enough that it **just barely** fits on one line in your screen.

Discuss with your group what this function is doing. How does it work? What do `i,j` really mean? What does it return? Turn in your debug function, and also the results of your group discussion.

Solution:

```
def debug_print(data, i,j):
    str_data = []
    for v in data:
        str_data.append(str(v))

    pivot = str_data[0]
    low   = " ".join(str_data[1:i   ])
    mid   = " ".join(str_data[  i:j ])
    hi    = " ".join(str_data[    j:])

    print(f"{pivot} ({low}) mid ({hi})")
```

Activity 4 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Analyze the performance of the `partition()` function from the previous activity (not counting your debug code) - how long does it take, in big-Oh notation? (Hint: how many times do `i,j` have to advance before the main `while` loop ends? How much work do you have to advance one of them by one step?)

Then, verify that your analysis is correct with experimentation.

Solution: This function takes $O(n)$ time, because it takes constant time to move `i,j` toward their destination - and we can only do $O(n)$ such steps.