1 Administrative Topics

- Please come to the candidate’s talk today at 4:15pm in Davis 201.
- I return the quizzes.

2 Sorting

One of the most common class of algorithms used in programming is the class of algorithms that sorts data. For example, in my project to make a public version of my calendar, I needed to sort my calendar events from earliest to latest. When we have large amounts of data, sorting efficiently is really important. Today, we will go over two sorting algorithms and apply them to lists of numbers. We will see that the first algorithm is inefficient and the second is much more efficient.

2.1 Bubble Sort

The idea behind bubble sort is that we allow the biggest values to “bubble up” to the top (end) of the list. First, we bubble up the biggest value, then the second largest, etc. We do so by iterating through the list and swapping pairs of values. I highly recommend Wikipedia for a visualization of the sort. Here, I include my Python code
```python
def bubble_sort(nums):
    # loop over all values
    for i in range(len(nums)):
        # bubble up the value
        for j in range(len(nums)-i-1):
            if nums[j] > nums[j+1]:
                tmp = nums[j]
                nums[j] = nums[j+1]
                nums[j+1] = tmp
```

To measure the time-efficiency of an algorithm, we perform what is called complexity analysis. We determine how much work needs to be done as a function of the number of elements in the list (N). For this algorithm, it means we want to determine how many times the inner loop body executes. It is

\[ N + (N-1) + (N-2) + \cdots + 1 = \frac{N(N-1)}{2} \]

In complexity analysis, we ignore constants (if they are just multiplying or adding). That means the time it takes is “on the order of” \( N^2 \), which we write as \( O(N) \). We will visit this in much more detail in CS231. It is called Big-Oh notation.

A \( O(N^2) \) sorting algorithm is not considered efficient. The best algorithms are \( O(N \log N) \).

### 2.2 Merge Sort

To get an \( O(N \log N) \) algorithm, it is often a good idea to use recursion - we divide and conquer. That is what merge sort does. The basic idea behind merge sort is that we keep subdividing the list into smaller and smaller parts.

We sort the smallest lists (list of length 1) and then merge pairs of lists. We do this recursively - in each function call, we make recursive calls to sort the first half of the list and then to sort the second half of the list, then we merge the two sorted halves.

Again, I highly recommend Wikipedia for a visualization of the merge sort working. Here is my code
def merge_sort(nums):
    if len(nums) < 2:
        # A list this short is ipso facto sorted.
        # return it.
        return nums
    midpt = len(nums)/2
    # Divide list into two smaller lists and
    # sort them
    nums1 = merge_sort(nums[0:midpt])
    nums2 = merge_sort(nums[midpt:]
    # Merge the two sorted lists
    sorted = []
    idx1 = 0
    idx2 = 0
    while idx1 < len(nums1) and idx2 < len(nums2):
        if nums1[idx1] < nums2[idx2]:
            sorted.append(nums1[idx1])
            idx1 += 1
        else:
            sorted.append(nums2[idx2])
            idx2 += 1
    if idx1 < len(nums1):
        sorted.extend(nums1[idx1:])
    elif idx2 < len(nums2):
        sorted.extend(nums2[idx2:])
    return sorted

The complexity analysis with recursion is not quite as straight-forward as it is with loops. We need to determine how many times the recursive calls will be made, and how much work is done at each call. Since we are splitting the list in 2 at each call, we can think of the lists as being organized into a binary tree. That binary tree has height $\log_2 N$. At each level of the tree, each number in the list is examined. So the total amount of work is $O(N \log N)$. 

3