Sorting

Sorting is to computing what the Empire State Building is to New York City—an iconic, inimitable problem that is ubiquitous in the field. So many problems become easier after sorting that it's usually the first and simplest strategy to try when encountering a new problem. For example, given a list of n numbers, you can find the median number using around n operations. Showing this is true is non-trivial. But imagine you sorted the numbers first. Now finding the median is as easy as picking grabbing the number occurring in the middle of the list. In most cases, sorting the numbers take more than n operations, but not much more. Here we will consider a general purpose sorting routine called MERGESORT that is recursive and has applications to sorting big data.

MERGESORT

Let's keep it simple: suppose you have a list of number L and you want to sort them in ascending order. Thinking inductively, imagine that you could solve the sorting problem on smaller lists. In other words, if you divide L into two lists L_1 and L_2 of size $\lceil L/2 \rceil$ and $\lceil L/2 \rceil$ respectively, then you can assume that L_1 and L_2 come back sorted. What remains is to merge L_1 and L_2 into a sorted version of L. Because L_1 and L_2 are sorted, this is quite natural: compare the first items in each list and insert the smaller into the new *sorted* list. Suppose the smaller item was in L_1. Now compare the second item of L_1 with the first item in L_2. Now suppose the first item of L_2 is smaller. Add it to the sorted list and continue the process until one of the lists is empty.

Here's some Python code to perform the MERGE step.

```
1  def merge(L1, L2):
2
3    i1 = i2 = 0
4    L = []
5    while i1 < len(L1) and i2 < len(L2):
6      if L1[i1] < L2[i2]:
7        L.append(L1[i1])
8        i1 +=1
9      else:
10       L.append(L2[i2])
11       i2 += 1
12   if i1 == len(L1):
13     L.extend(L2[i2:])
14   else:
15     L.extend(L1[i1:])
16   return L
```

With MERGE in hand, writing MERGESORT amounts to a few small lines of code:

```
1  def mergesort(L):
2    if len(L) < 2:
3      return L
4    else:
5      mid = len(L) // 2
6      return merge(mergesort(L[0:mid]), mergesort(L[mid:]))
```

Efficiency of MERGESORT

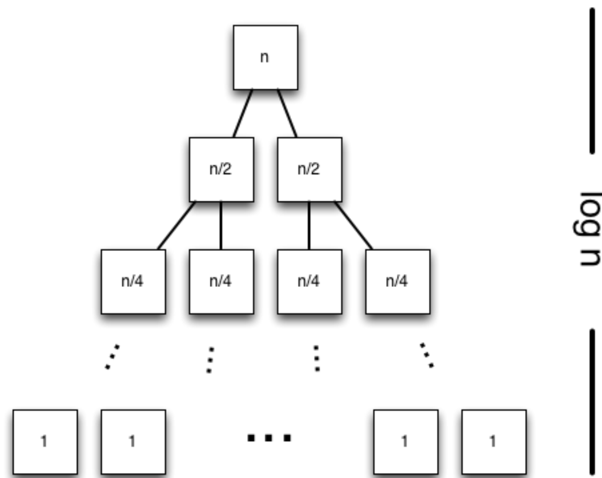
The primary unit of work in classical sorting is the comparison. Counting comparisons allow us, with a slight bit of irony, *compare* sorting algorithms. When we compare two elements, we establish their relative ordering. It's easy

to see that we could always perform $\binom{n}{2} \approx n^2$ comparisons to establish a total ordering. A natural question is, *can we do better?* Let's take a stab at counting the number of comparisons that MERGESORT performs in the worst case starting with a list of n items.

Let T_n be the largest number of comparisons performed by MERGESORT on a list of n numbers. Mathematically we have

$$T_n = \begin{cases} 2T(n/2) + n & (n > 1) \\ 1 & \text{(otherwise.)} \end{cases}$$

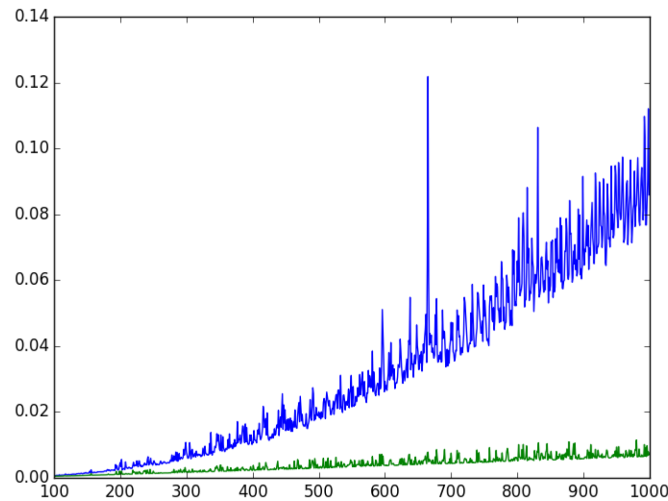
Let's get a sense of this by drawing a picture. Each node represents a call to MERGESORT and shows the number of comparisons performed. Each level of the recursion has n total comparisons. There are $\log n$ levels. So the total number of comparisons is $n \log n$.

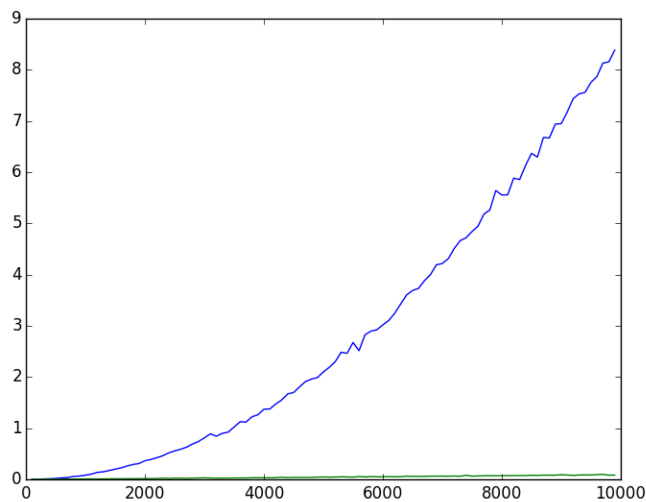


Here is an algorithm called INSERTIONSORT that works as follows: it considers the element at position i under the invariant that the list from positions 0 to $i-1$ are already sorted. It then inserts the element at position i in the correct spot by moving backward and swapping elements. Here is the Python code.

```
1 def insertionsort(L):
2     for i in range(len(L)):
3         for j in range(i−1,−1,−1):
4             if L[j+1] < L[j]:
5                 L[j], L[j+1] = L[j+1], L[j]
6             else:
7                 break
8     return L
```

Below are two plots that show the running time (y axis) of MERGESORT (green) and INSERTIONSORT (blue) on random data of size $100 \leq n \leq 1000$ and $100 \leq 10000$ by 100 (x-axis).





Adding `reverse` and `key` options

The builtin Python sorting routine accepts two optional arguments: `key` and `reverse`. The `key` is a function applied to the data before it's compared. The `reverse` option signals that the data should be sorted in non-increasing, rather than non-decreasing, order.

To implement the key option in `merge`, we just add a new argument `key` and apply the function before the comparisons.

```
1  def merge(L1, L2, key):
2
3      i1 = i2 = 0
4      L = []
5      while i1 < len(L1) and i2 < len(L2):
6          if key(L1[i1]) < key(L2[i2]):
7              L.append(L1[i1])
8              i1 +=1
9          else:
10             L.append(L2[i2])
11             i2 += 1
12     if i1 == len(L1):
13         L.extend(L2[i2:])
14     else:
15         L.extend(L1[i1:])
16     return L
```

To implement `reverse` we isolate our recursive procedure in a helper function and reverse the results if desired.

```
1  def mergesort(L, key=None, reverse=False):
2
3      if key is None:
4          key = lambda x: x
5
6      def helper(L):
7          if len(L) < 2:
8              return L
9          else:
10             mid = len(L) // 2
11             return merge(helper(L[0:mid]), helper(L[mid:]), key)
12
13     L = helper(L)
14     if reverse:
15         L.reverse()
16     return L
```