

Linear versus Binary Search

Please see the slides linked here. If you're not spending two or three hours on this, you're selling yourself short. First, we will review the code we generated in yesterday's class. We looked at three different ways of finding an element in a lyst. Specifically, we wrote three different functions to find the index of the first occurrence of **key** in the list **lyst**. Further, we counted the number of comparisons made by all three functions. The below are slightly modified to avoid the redundancy in code we wrote earlier.

- **findLinear(lyst, key)**: a linear search making no assumptions on the order of elements in **lyst**.

```
def findLinear(lyst, key):
    count = 0
    index = -1
    for i in range(len(lyst)):
        count += 1
        if lyst[i] == key:
            index = i
            break

    print("%d comparisons made, to find at index %d." \
          % (count, index))
    return index
```

- **findLinearSorted(lyst, key)**: a linear search making the assumption that **lyst** is sorted. This allows us to potentially quit early, as in when we compare against an element greater than **key** (as **key** cannot be found at any later position).

```
def findLinearSorted(lyst, key):
    count = 0
    index = -1
    for i in range(len(lyst)):
        count += 1
        if lyst[i] == key:
            index = i
```

```

        break
    else:
        count += 1
        if lyst[i] > key:
            break

    print("%d comparisons made, to find at index %d." \
          % (count, index))
    return index

```

- **binSearch(lyst, key)**: the “guessing game”-like process where the list is assumed in order. We first compare against the middle element in list, which allows us to eliminate half of the list (if the element compared against is not the key).

```

def binSearch(lyst, key):
    lo = 0
    hi = len(lyst) - 1
    count = 1
    index = -1
    while lo <= hi:
        mid = (lo + hi)//2
        count += 1
        if lyst[mid] == key:
            index = mid
            break
        else:
            count += 1
            if lyst[mid] < key:
                lo = mid + 1
            else:
                hi = mid - 1
    count += 1 #about to compare lo <= hi again

    print("%d comparisons made, to find at index %d." \
          % (count, index))
    return index

```

We now generate a list to test these search functions and observe their counts for keys either in the list or not and where. We’ll have the list contain 1M elements, in

the range 1 to 10M. Further, we'll eliminate some element, so we can also look for a key that is not in the list.

```
>>> import random
>>> lyst = sorted([random.randint(1,10000000) for i in range(1000000)])
>>> while lyst.count(750000):
        lyst.remove(750000)
>>> findLinear(lyst, 750000)
1000000 comparisons made, to find at index -1.
-1
>>> findLinearSorted(lyst, 750000)
150480 comparisons made, to find at index -1.
-1
#We can quit early on an element not in the list using findLinearSorted
>>> binSearch(lyst, 750000)
61 comparisons made, to find at index -1.
-1
#binSearch wins! What if we look for the last element?
>>> mx = lyst[-1]
>>> findLinear(lyst, mx)
1000000 comparisons made, to find at index 999999.
999999
>>> findLinearSorted(lyst, mx)
1999999 comparisons made, to find at index 999999.
999999
>>> binSearch(lyst, mx)
59 comparisons made, to find at index 999999.
999999
```

So binary search provides an intense savings in many scenarios. We can see this formally in the same way we looked at the guessing game before. By reducing the number of possibilities by half for *each* iteration, there can only be so many iterations possible, namely approximately $\log(n)$ for n the number of possibilities.

Please go to the next page.

Selection Sorting

Please see the slides linked here (also on the course schedule). Please spend some time before coding actually trying to sort a few playing cards. Remember this class is about things you already know how to do. But the real issue is whether you can describe exactly what you are doing.

Sort 5+ playing cards. Exactly how do you do it? Write the english instructions (pseudo-code) down.

What you may have done is fairly obvious. Find the smallest card in the hand, take it out of the hand, and place it on the counter (in a separate pile). If this isn't what you did, keep reading anyway.

Let's convert this to code. We'll have a list of integers. We start a new list, and simply append to the new list the pop of the minimum from the first list. We will write a function `easySort`, that accepts a `lyst` and returns a new sorted list in increasing order.

```
def easySort(lyst):
    r = []
    while len(lyst) > 0:
        r.append(lyst.pop(lyst.index(min(lyst))))
    return r
```

You can see we simply find the index of the minimum (`lyst.index(min(lyst))`), then pop it from the list and append to the new list `r`. We do this until `lyst` is empty. This is broadly a **SELECTION** sorting algorithm. We are *selecting* the minimum element and placing it in a new list.

```
>>> import random
>>> lyst = [random.randint(1,100) for i in range(10)]
>>> lyst
[81, 71, 86, 38, 15, 39, 14, 89, 70, 75]
>>> easySort(lyst)
[14, 15, 38, 39, 70, 71, 75, 81, 86, 89]
>>> lyst
[]
```

At the cost of more complicated code, we can write this in a way that actually sorts the provided `lyst`, rather than destroy it as above. Rather than selecting the minimum element and placing it in a new list, we will select the minimum element in the `lyst` and then place it in the first (0^{th}) index. What was previously in the 0^{th}

index we will place where that minimum was (a.k.a, we'll swap the two elements). Then we simply do the same thing again on the rest of the list (not including the first element). Let's see the `swap` function

```
def swap(lyst, i, j):
    temp = lyst[i]
    lyst[i] = lyst[j]
    lyst[j] = temp
```

A subpiece of this program is one we've written before, finding the index of the minimum element in a list. We'll modify it slightly so that we don't have to start at the 0th index, but can start anywhere.

```
def selectionSort(lyst):
    for i in range(len(lyst)):
        #find the index of the min starting at i
        mi = i
        for j in range(i, len(lyst)):
            if lyst[j] < lyst[mi]:
                mi = j
        #Now mi is the index where the minimum is
        #simply swap
        swap(lyst, i, mi)
```

```
>>> import random
>>> lyst = [random.randint(1,100) for i in range(10)]
>>> lyst
[33, 78, 80, 9, 46, 69, 90, 70, 90, 59]
>>> selectionSort(lyst)
>>> lyst
[9, 33, 46, 59, 69, 70, 78, 80, 90, 90]
```

Notice that our `selectionSort` function does not actually return anything. It sorts the list *in place*, and sorts the list as a side effect of the function.

Let's consider the complexity of the algorithm `selectionSort`. The comparison `lyst[j] < lyst[mi]` is inside of two `for` loops now. We can see the outer loop executes a number of times equal to the length of the input `lyst`. So, n times we do the inner loop (which is responsible for finding the index of the minimum element in the list).

Next we consider how many times the inner loop executes for each of the n iterations of the outer loop. Well, the first time we are looking through the entire list for

the index of the minimum. So that's n . The second time we execute the inner loop, we are searching through the entire list except for the first element (in which we will have already placed the minimum). So, that's $n - 1$ more comparisons. Adding all these up, we have a sum to n , which we've seen:

$$n + (n - 1) + (n - 2) + \dots + 1 = \frac{n * (n + 1)}{2} = \frac{n^2}{2} + \frac{n}{2} \approx O(n^2) \quad (1)$$

As n grows large, the term $\frac{n^2}{2}$ becomes very much larger than the $\frac{n}{2}$ term. For example, for $n = 1000$, the first term is 500000 while the second term is relatively tiny at 500. Thus we ignore the second term in categorizing the complexity of the algorithm. Further, the factor $1/2$ applied to the n^2 in the first term is also not entirely important (again, as n grows large). Thus we conclude that the `selectionSort` algorithm has complexity $O(n^2)$ (big-O complexity).

Insertion Sorting

Another way you may have sorted the list is as follows: Take one card and place it separately. Then take the next card, and place it either before or after the first based on a comparison. Then take the next card and **INSERT** it into the two-card list you've built. Then continue to place each new card where it belongs in the separate list you're constructing. Let's see this in code (which is a little harder to write):

```
def easySort2(lyst):
    r = []
    while len(lyst) > 0:
        x = lyst.pop()
        #Find where x belongs in r.
        i = 0
        while i < len(r) and x > r[i]:
            i += 1
        #i, started at 0, is now where in r, x belongs
        #so let's insert it there.
        r.insert(i, x)
    return r
```

Now, while the code above is functional, it does again destroy the original list. Let us instead write an **INSERTION** sort function that, like the `selectionSort` function, sorts the list as a side effect. We will consider the list in two parts: the

first part, which is already sorted (this starts out with only a single element in it), and the second part, which contains the elements yet to be sorted. n times, we will *insert* an element from the second part into the first. Finally, in order to keep the list in place, we must *shift* elements over to make space for the element we're trying to insert. You can see this physically if you execute the scheme while also ensuring that there are only fixed locations for the cards.

```
def insertionSort(lyst):
    #Start with second element, insert it to the left.
    for i in range(1, len(lyst)):
        t = lyst[i] #the element to insert
        j = i-1
        while j >= 0 and t < lyst[j]:
            lyst[j+1] = lyst[j]
            #Above, we shift the element over to make
            #space for t.
            j -= 1
        lyst[j+1] = t

>>> import random
>>> lyst = [random.randint(1,100) for i in range(10)]
>>> lyst
[32, 12, 89, 56, 57, 34, 18, 91, 2, 10]
>>> insertionSort(lyst)
>>> lyst
[2, 10, 12, 18, 32, 34, 56, 57, 89, 91]
```

If we consider the complexity of this scheme, we would want to count the number of times the comparison `t < lyst[j]` takes place. In that it is part of an indefinite loop, we cannot be sure how many times it will execute. If the list is already sorted, then only one comparison will have to be executed. If the list is in reverse order, then each new element that we insert must be compared to all the elements prior. One thing perhaps we can see though, is that this algorithm at worst will do only as many comparisons as `selectionSort` above, which must search the entire rest of the list for each iteration. Thus, we will argue that best case, `insertionSort` is $O(n)$, whereas average and worst case we will claim $O(n^2)$.

On Wednesday, we will consider these issues further, and through print statements inside the code we will look at the specific mechanics of these algorithms. Thank you.