Overview of Order of Growth

Why do we care? When Twitter started out with 10 users, it probably wanted to write code that would be able to scale to 10 billion users. So we want to know how much work a computer will do given a piece of code. Twitter engineers had to figure out: if we had 10 billion users instead of 10 users, how much more time/computation would it take?

- Below is a conceptual example that is not exact but is just intended to give you a flavor of orders of growth. We'll make it more concrete later in this recitation.
- For simplicity let's just suppose Twitter wants to scale from 10 users to 100 users.
- Suppose all Twitter ever does is start up a web server, but doesn't show any webpages to anyone. The amount of work it does would always be the same no matter how many users you have. (concept: “constant time”)
  - If this sounds too unrealistic or cheesy, that’s a good hunch! Most real-world problems don’t have constant-time solutions unless they’re really clever.
- Now suppose all Twitter ever does is show one profile page per person, and that’s it. So the amount of work is just proportional to the # users, so the difference in work would be 100 - 10 = 90 units of work (concept: “linear time”)
- But what if Twitter wants to check relationships between two people? Then the amount of work is proportional to the # users squared, because there are n^2 possible relationships. So difference in work would be 100^2 - 10^2 = 9900. This is a much bigger difference!! (concept: “polynomial time”)
- What about if the amount of work is proportional to 2^(# users)? So difference in work would be 2^100 - 2^10 = gigantic number (concept: “exponential time”)

- Normally, we want to think about big, big inputs (think “as n approaches infinity, what happens to the amount of work?”)

- big O notation
  - big O usually refers to the upper bound, or worst case scenario (actual runtime can be faster)
  - definition:
    - let amount of work = f(n)
    - f(n) “is in” O(g(n)) if for sufficiently large n, f(n) <= cg(n), where c is a positive real number
  - ignore lower-order terms and constant factors
    - O(n^3 + n^2 + n + 1) → O(n^3). We just care about the n^3 term since it
sort of masks everything else.

- $O(1000000000n) \rightarrow O(n)$

- Some things that take constant time $O(1)$
  - Basic operations, such as +,-,*,= or comparisons
  - certain dictionary operations:
    - i.e. `mydict[‘thekey’]` #getting value given a key
    - i.e. `if ‘thekey’ in mydict:` #checking if a key is in a dictionary

- Common orders of growth, from best to worst:
  - $O(1)$: constant time, takes the same amount of time regardless of input size
  - $O(\log n)$: logarithmic time
  - $O(n)$: linear time
  - $O(n\log n)$
  - $O(n^2), O(n^3), O(n^4)$ etc.: polynomial time
  - $O(2^n)$: exponential time (these are pretty bad)

- Strategies: think about iterations of a loop or recursive calls.
  - 1) Focus on loops:
    - # of loops that happen, given input $n$
    - amount of work each loop does
    - # loops x work/loop = total work
  - 2) Focus on recursions:
    - # of recursive calls that will be made, given input $n$
    - amount of work each recursive call does
    - # recursive calls x work/call = total work
    - If you find that the work per recursive call varies depending on which recursive call it is, then you could try strategy 3) instead
  - 3) Another way to evaluate recursive functions:
    - total work = height of the recursion tree * work/tree-level

---

**Order of Growth Analysis**

1) Simple example

```python
def iter(n):
    b = 10
    for i in range(n):
        b+=1
    return b
```

- How many loops? $O(n)$
- How much work per loop? $O(1)$
- Total work: $n \times 1 \rightarrow O(n)$
- notice that $b$ is irrelevant to the # of loops

2) Intersection of two arrays, assuming both arrays are length order $n$

```python
def intersect(l1, l2):
    intersection = []
    for i in l1:
        for j in l2:
            if i == j:
                intersection.append(i)  # assume appending takes constant time
    return intersection
```

- How many outer loops? $O(\text{len}(l1))$
- How many inner loops? $O(\text{len}(l2)) = O(\text{len}(l1))$ given our assumption that the two lists are similar in size
- How much work per loop? $O(1)$ because appending happens in constant time
- Total work $= n \times n \rightarrow O(\text{len}(l1)^2)$

2)

```python
def beep(n):
    sum = 0
    while n >= 2:
        sum += n
        n = n / 2
    return sum
```

- How many loops?
  n gets halved each iteration of the while loop.
  i.e. if $n$ starts out at 8, $n$ becomes 8, 4, 2, 1, etc.
  So there are $\log$-base-two-of-$n$ loops. $\log_2(n)$
- How much work per loop? $O(1)$
- Total time: $\log n \times 1 \rightarrow O(\log n)$ When talking about complexity, we just say $O(\log n)$ without saying the exact base.
- Notice that the sum+=$n$ is mostly a distraction, since # loops doesn’t depend on sum

-----------------------------------

More Complicated Exercises!

1)
def bop(n):
    for i in range(5):
        n = n/2
    return n

- How many loops? O(1) Notice that # loops does not depend on n. No matter what n is, the most number of loops we ever do is 5.
- How much work per loop? O(1) since we’re just doing division
- O(1)

2)
def bap(n):
    if n <= 1:
        return 0
    else:
        return bap(n-3) + bap(n-1)

- Draw the recursive tree!
- How many recursive calls? 2 + 2^2 + 2^3 + 2^4 + 2^5 etc…. + 2^n = O(2^n)
    - tree depth is n
- How much work in each recursive call? O(1), just adding two things
- Total work: O(2^n)

3) Intersection of two arrays using dictionaries
***Reminder before starting***: many dictionary operations are constant time (i.e. checking if a key is in dictionary, finding value given key)

#assume lengths of l1 and l2 are similar
def intersect_better(l1, l2):
    seen = {}
    intersection = []
    for i in l1:
        seen[i] = “yes”
    for j in l2:
        if j in seen:
            # assume appending takes constant time
            intersection.append(j)
    return intersection

- How many loops? “i” loop is O(len(l1)) + “j” loop is O(len(l2)) = O(2*len(l1)) = O(len(l1)) loops
    Notice that the loops are not nested, so we can’t multiply them like we often do with double for loops.
- In each loop, how much work is done? O(1)
- total work = len(l1) x 1 → O(len(l1))

4)
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

def sum_of_factorial(n):
    if n == 0:
        return 1
    else:
        return factorial(n) + sum_of_factorial(n-1)

factorial:
- how many recursive calls? O(n)
- how much work per call? O(1)
- total work = n x 1 = O(n)

sum_of_factorial:
- how many recursive calls to sum_of_factorial? O(n)
- how much work per call?
  - first call to factorial() does O(n) because it calls factorial(n)
  - second call to factorial() does O(n-1) because it calls factorial(n-1)
  - etc.
  - eventually does O(1) because calls factorial(0)
  - so average work per sum_of_factorial recursive call is O(n/2) = O(n)
- total work = n x n = O(n^2)

-------------------------------

**Sorting Algorithms**

**Selection sort** (n^2)
- at each step, prefix is sorted, suffix is unsorted
- keep lengthening the prefix and shortening the suffix
- at the end of each loop, the min element from suffix becomes the end of prefix
how many outer loops? $O(n)$
how many inner loops? $O(n/2)$
  - first time: $i$ takes $n$ steps
  - second time: $i$ takes $n-1$ steps
  - third time: $i$ takes $n-2$ steps
  - etc.
  - so on average, $i$ takes $n/2$ steps (in other words, $n/2$ loops)
- total work = $n \times n/2 \rightarrow O((n^2)/2) \rightarrow O(n^2)$ # because we ignore constants

```python
def selSort(L):
    """Assumes that L is a list of elements that can be compared
    using >. Sorts L in ascending order""
    prefixEnd = 0
    print 'List to be sorted =', L
    while prefixEnd != len(L):
        # look at each element in suffix
        for i in range(prefixEnd, len(L)):
            if L[i] < L[prefixEnd]:
                # swap position of elements
                L[prefixEnd], L[i], L[prefixEnd]
        print 'Partially sorted list =', L
        prefixEnd += 1
```

Merge sort (nlogn)

1) break list in half
2) recursively sort both halves
3) merge the sorted halves
- Draw out of the tree of list(s) of numbers that are being processed at every level of recursion
- How many levels of the recursive tree are there? O(logn)
- How much computation at each level of the tree? O(n) -- each merge takes O(n) time because the lists are already sorted
  - Why does merging take only O(n) time? When both lists are already sorted, we only need to look at each item once in order to merge them into one big sorted list. It would take longer than O(n) if the two lists weren't already sorted. *This is the beauty of mergesort! It enables each level of recursion to merge in O(n) time, because the level before it fed in two sorted lists.*
- Total time: n * logn = O(nlogn)
- QUESTION: why couldn’t we just easily use our strategy mentioned earlier of doing # recursive calls x work/call?
  Answer: because in this case, work/call varies depending on which recursive call we’re on. We could try to figure out the average work/call, but a slightly simpler different strategy is to do it in the way we have above (what we did was # levels x work/level).

```python
def mergeSort(L, compare = operator.lt):
    """Assumes L is a list, compare defines an ordering on elements of L
    Returns a new sorted list containing the same elements as L""
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = mergeSort(L[:middle], compare)
        right = mergeSort(L[middle:], compare)
        return merge(left, right, compare)
```

**Exercise**

How does this mysterious sorting function work?
What’s the complexity?
def mystery_sort(L):
    for i in xrange(1, len(L)):
        j = i
        while L[j-1] > L[j] and j > 0:
            L[j-1], L[j] = L[j], L[j-1]
            j -= 1
    return L

Explanation:
- This algorithm is called insertion sort:
  - keep prefix sorted and suffix unsorted
  - swap end-of-prefix leftward until it is in the correctly sorted location (the j loop)
  - repeat incrementing the end-of-prefix location (the i loop)

- How many outer loops? $O(len(L))$
- How many inner loops?
  - First time, $i=1$, so you might have to do 1 swap (1 loop)
  - Second time, $i=2$ so you might have to do 2 swaps
  - Third time, $i=3$, so you might have to do 3 swaps
  - Eventually, you might have to do $(n-1)$ swaps
  - Average # inner loops is $len(L)/2$
- So total is $len(L) \times len(L)/2 = O(len(L)^2)$