COMPLEXITY CLASSES
EXAMPLES
(download slides and .py files to follow along)
6.100L Lecture 23
Ana Bell
Theta

- **Theta Θ** is how we denote the *asymptotic complexity*

- We look at the **input term that dominates** the function
  - Drop other pieces that don’t have the fastest growth
  - Drop additive constants
  - Drop multiplicative constants

- End up with only a **few classes of algorithms**
- We will look at code that lands in each of these classes today
WHERE DOES THE FUNCTION COME FROM?

- Given code, start with the input parameters. What are they?
- Come up with the equation relating input to number of ops.
  - \( f = 1 + \text{len}(L1) \times 5 + 1 + \text{len}(L2) \times 5 + 2 = 5 \times \text{len}(L1) + 5 \times \text{len}(L2) + 3 \)
  - If lengths are the same, \( f = 10 \times \text{len}(L) + 3 \)
- \( \Theta(f) = \Theta(10 \times \text{len}(L) + 3) = \Theta(\text{len}(L)) \)

```python
def f(L, L1, L2):
    inL1 = False
    for i in range(len(L1)):
        if L[i] == L1[i]:
            inL1 = True
    inL2 = False
    for i in range(len(L2)):
        if L[i] == L2[i]:
            inL2 = True
    return inL1 and inL2
```
WHERE DOES THE FUNCTION COME FROM?

- A quicker way: no need to come up with the exact formula. Look for loops and anything that repeats wrt the input parameters. Everything else is constant.

```python
def f(L, L1, L2):
    inL1 = False
    for i in range(len(L1)):
        if L[i] == L1[i]:
            inL1 = True
    inL2 = False
    for i in range(len(L2)):
        if L[i] == L2[i]:
            inL2 = True
    return inL1 and inL2
```
COMPLEXITY CLASSES

n is the input

We want to design algorithms that are as close to top of this hierarchy as possible

- \( \Theta(1) \) denotes constant running time
- \( \Theta(\log n) \) denotes logarithmic running time
- \( \Theta(n) \) denotes linear running time
- \( \Theta(n \log n) \) denotes log-linear running time
- \( \Theta(n^c) \) denotes polynomial running time
  (c is a constant)
- \( \Theta(c^n) \) denotes exponential running time
  (c is a constant raised to a power based on input size)
CONSTANT COMPLEXITY
CONSTANT COMPLEXITY

- Complexity **independent of inputs**
- Very few interesting algorithms in this class, but can often have pieces that fit this class
- **Can have loops or recursive calls**, but number of iterations or calls independent of size of input
- Some built-in operations to a language are constant
  - Python indexing into a list \( L[i] \)
  - Python list append \( L.append() \)
  - Python dictionary lookup \( d[key] \)
CONSTANT COMPLEXITY: EXAMPLE 1

```python
def add(x, y):
    return x + y
```

- Complexity in terms of either x or y: $\Theta(1)$
CONSTANT COMPLEXITY: EXAMPLE 2

```python
def convert_to_km(m):
    return m*1.609
```

- Complexity in terms of m: $\Theta(1)$
CONSTANT COMPLEXITY: EXAMPLE 3

def loop(x):
    y = 100
    total = 0
    for i in range(y):
        total += x
    return total

- Complexity in terms of x (the input parameter): $\Theta(1)$
LINEAR COMPLEXITY
LINEAR COMPLEXITY

- Simple **iterative loop** algorithms
  - Loops must be a **function of input**
- Linear search a list to see if an element is present
- Recursive functions with **one recursive call and constant overhead** for call
- Some built-in operations are linear
  - `e in L`
  - Subset of list: e.g. `L[:len(L)//2]`
  - `L1 == L2`
  - `del(L[5])`
COMPLEXITY EXAMPLE 0
(with a twist)

- Multiply x by y

```python
def mul(x, y):
    tot = 0
    for i in range(y):
        tot += x
    return tot
```

- Complexity in terms of y: $\Theta(y)$
- Complexity in terms of x: $\Theta(1)$
BIG IDEA

Be careful about what the inputs are.
LINEAR COMPLEXITY: EXAMPLE 1

- Add characters of a string, assumed to be composed of decimal digits

```python
def add_digits(s):
    val = 0
    for c in s:
        val += int(c)
    return val
```

- $\Theta(len(s))$
- $\Theta(n)$ where $n$ is $len(s)$

Loop goes through $len(s)$ times: $\Theta(len(s))$
Everything else is constant. $\Theta(1)$
LINEAR COMPLEXITY: EXAMPLE 2

- Loop to find the factorial of a number \( \geq 2 \)
  ```python
def fact_iter(n):
    prod = 1
    for i in range(2, n+1):
      prod *= i
    return prod
  ```
  Loop goes through \( n-1 \) times:
  \( \Theta(n) \)
  Everything else is constant.
  \( \Theta(1) \)

- Number of times around loop is \( n-1 \)
- Number of operations inside loop is a constant
  - Independent of \( n \)
- Overall just \( \Theta(n) \)
**FUNNY THING ABOUT FACTORIAL AND PYTHON**

- Eventually grows faster than linear
- Because Python increases the size of integers, which yields more costly operations
- For this class: ignore such effects

```python
iter fact(40) took 3.10e-06 sec (322,580.65/sec)
iter fact(80) took 6.00e-06 sec (166,666.67/sec)
iter fact(160) took 1.34e-05 sec (74,626.87/sec)
iter fact(320) took 3.39e-05 sec (29,498.53/sec)
iter fact(640) took 1.18e-04 sec (8,488.96/sec)
iter fact(1280) took 4.31e-04 sec (2,322.88/sec)
iter fact(2560) took 1.33e-03 sec (752.73/sec)
iter fact(5120) took 4.94e-03 sec (202.24/sec)
iter fact(10240) took 1.90e-02 sec (52.50/sec)
iter fact(20480) took 7.66e-02 sec (13.06/sec)
iter fact(40960) took 3.35e-01 sec (2.99/sec)
iter fact(81920) took 1.60e+00 sec (0.62/sec)
```
LINEAR COMPLEXITY: EXAMPLE 3

```python
def fact_recur(n):
    """ assume n >= 0 """
    if n <= 1:
        return 1
    else:
        return n*fact_recur(n - 1)
```

- Computes factorial recursively
- If you time it, notice that it runs a bit slower than iterative version due to function calls
- $\Theta(n)$ because the number of function calls is linear in n
- **Iterative and recursive factorial** implementations are the same order of growth

Think about the function call stack: $\Theta(n)$

Everything else is constant.

$\Theta(1)$
LINEAR COMPLEXITY: EXAMPLE 4

```python
def compound(invest, interest, n_months):
    total = 0
    for i in range(n_months):
        total = total * interest + invest
    return total
```

- \( \Theta(1) \times \Theta(n\_months) = \Theta(n\_months) \)
- \( \Theta(n) \) where \( n=n\_months \)

- If I was being thorough, then need to account for assignment and return statements:
  - \( \Theta(1) + 4 \times \Theta(n) + \Theta(1) = \Theta(1 + 4n + 1) = \Theta(n) \) where \( n=n\_months \)
COMPLEXITY OF ITERATIVE FIBONACCI

```python
def fib_iter(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_i = 0
        fib_ii = 1
        for i in range(n-1):
            tmp = fib_i
            fib_i = fib_ii
            fib_ii = tmp + fib_ii
        return fib_ii
```

$$\Theta(1) + \Theta(1) + \Theta(n) * \Theta(1) + \Theta(1) \Rightarrow \Theta(n)$$
POLYNOMIAL COMPLEXITY
POLYNOMIAL COMPLEXITY (OFTEN QUADRATIC)

- Most **common polynomial algorithms are quadratic**, i.e., complexity grows with square of size of input
- Commonly occurs when we have **nested loops** or recursive function calls
QUADRATIC COMPLEXITY: EXAMPLE 1

```python
def g(n):
    """ assume n >= 0 """
    x = 0
    for i in range(n):
        for j in range(n):
            x += 1
    return x
```

- Computes $n^2$ very inefficiently
- Look at the loops. Are they in terms of the input?
  - Nested loops
  - Look at the ranges
  - Each iterating $n$ times
- $\Theta(n) \times \Theta(n) \times \Theta(1) = \Theta(n^2)$
QUADRATIC COMPLEXITY: EXAMPLE 2

- Decide if L1 is a subset of L2: are all elements of L1 in L2?

  Yes:
  
  L1 = [3, 5, 2]  
  L2 = [2, 3, 5, 9]

  No:
  
  L1 = [3, 5, 2]  
  L2 = [2, 5, 9]

```python
def is_subset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```
def is_subset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True

Outer loop executed \( \text{len}(L1) \) times
Each iteration will execute inner loop up to \( \text{len}(L2) \) times
\( \Theta(\text{len}(L1) \times \text{len}(L2)) \)

If \( L1 \) and \( L2 \) same length and none of elements of \( L1 \) in \( L2 \)
\( \Theta(\text{len}(L1)^2) \)
Find intersection of two lists, return a list with each element appearing only once

Example:
L1 = [3, 5, 2]
L2 = [2, 3, 5, 9]
returns [2, 3, 5]

L1 = [7, 7, 7]
L2 = [7, 7, 7]
returns [7]

```python
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    unique = []
    for e in tmp:
        if not(e in unique):
            unique.append(e)
    return unique
```

Build the list with common elements in L1 and L2. May have dups
Keep only unique values
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    unique = []
    for e in tmp:
        if not(e in unique):
            unique.append(e)
    return unique

First nested loop takes $\Theta(len(L1) \times len(L2))$ steps.

Second loop takes at most $\Theta(len(L1) \times len(L2))$ steps. Typically not this bad.

- E.g: [7,7,7] and [7,7,7] makes tmp=[7,7,7,7,7,7,7,7,7]

Overall $\Theta(len(L1) \times len(L2))$
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt( (p1[0]-p2[0])**2 + (p1[1]-p2[1])**2 )
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist

len(L) * len(L)/2 iterations = len(L)^2 / 2

Θ(len(L)^2)
YOU TRY IT!

```python
def all_digits(nums):
    """ nums is a list of numbers """
    digits = [0,1,2,3,4,5,6,7,8,9]
    for i in nums:
        isin = False
        for j in digits:
            if i == j:
                isin = True
                break
        if not isin:
            return False
    return True
```

**ANSWER:**
What’s the input?
Outer for loop is $\Theta(\text{nums})$.
Inner for loop is $\Theta(1)$.
Overall: $\Theta(\text{len(nums)})$
YOU TRY IT!

- Asymptotic complexity of f? And if L1, L2, L3 are same length?

```python
def f(L1, L2, L3):
    for e1 in L1:
        for e2 in L2:
            if e1 in L3 and e2 in L3:
                return True
    return False
```

**ANSWER:**

\[ \Theta(\text{len}(L1)) \times \Theta(\text{len}(L2)) \times \Theta(\text{len}(L3) + \text{len}(L3)) \]

Overall: \( \Theta(\text{len}(L1) \times \text{len}(L2) \times \text{len}(L3)) \)

Overall if lists equal length: \( \Theta(\text{len}(L1)^2) \)
EXPONENTIAL COMPLEXITY
EXPONENTIAL COMPLEXITY

- Recursive functions where have more than one recursive call for each size of problem
  - Fibonacci
- Many important problems are inherently exponential
  - Unfortunate, as cost can be high
  - Will lead us to consider approximate solutions more quickly

\[
2^{30} \approx 1 \text{ million}
\]
\[
2^{100} > \# \text{ cycles than all the computers in the world working for all of recorded history could complete}
\]
COMPLEXITY OF RECURSIVE FIBONACCI

```python
def fib_recur(n):
    """assumes n an int >= 0 """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

- Worst case:
  \[ \Theta(2^n) \]
COMPLEXITY OF RECURSIVE FIBONACCI

- Can do a bit better than $2^n$ since tree thins out to the right
- But complexity is still order exponential
EXPONENTIAL COMPLEXITY: GENERATE SUBSETS

- Input is [1, 2, 3]
- Output is all combinations of elements of all lengths
  [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]

```python
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small + extra)
    return smaller + new
```

Base case: reach list of empty list
Create a list of just last element
All subsets without last element
For all smaller solutions, add one with last element
Combine those with last element and those without
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small + extra)
    return smaller + new
VISUALIZING the ALGORITHM

```
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small + extra)
    return smaller + new
```
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small + extra)
    return smaller + new
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small + extra)
    return smaller + new
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new

- Assuming append is constant time
- Time to make sublists includes time to solve smaller problem, and time needed to make a copy of all elements in smaller problem
EXPONENTIAL COMPLEXITY
GENERATE SUBSETS

def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new

- Think about size of smaller
  - For a set of size $k$ there are $2^k$ cases, doubling the size every call
  - So to solve need $2^{n-1} + 2^{n-2} + \ldots + 2^0$ steps $= \Theta(2^n)$

- Time to make a copy of smaller
  - Concatenation isn’t constant
  - $\Theta(n)$

- Overall complexity is $\Theta(n*2^n)$ where $n=len(L)$
LOGARITHMIC COMPLEXITY
def digit_add(n):
    """ assume n an int >= 0 """
    answer = 0
    s = str(n)
    for c in s[::-1]:
        answer += int(c)
    return answer

- Adds digits of a number together
  - n = 83, but the loop only iterates 2 times. Relationship?
  - n = 4271, but the loop only iterates 4 times! Relationship??

4 2 7 1
1

First time through loop, extract the least significant digit
def digit_add(n):
    """ assume n an int >= 0 """
    answer = 0
    s = str(n)
    for c in s[::-1]:
        answer += int(c)
    return answer

- Adds digits of a number together
  - n = 83, but the loop only iterates 2 times. Relationship?
  - n = 4271, but the loop only iterates 4 times! Relationship??
def digit_add(n):
    """ assume n an int >= 0 """
    answer = 0
    s = str(n)
    for c in s[::-1]:
        answer += int(c)
    return answer

- Adds digits of a number together
  - n = 83, but the loop only iterates 2 times. Relationship?
  - n = 4271, but the loop only iterates 4 times! Relationship??
def digit_add(n):
    """ assume n an int >= 0 """
    answer = 0
    s = str(n)
    for c in s[::-1]:
        answer += int(c)
    return answer

- Adds digits of a number together
  - n = 83, but the loop only iterates 2 times. Relationship?
  - n = 4271, but the loop only iterates 4 times! Relationship??

Last time through loop, extract the next least significant digit
def digit_add(n):
    """ assume n an int >= 0 """
    answer = 0
    s = str(n)
    for c in s[::-1]:
        answer += int(c)
    return answer

- Adds digits of a number together
- Tricky part: iterate over length of string, not magnitude of n
  - Think of it like dividing n by 10 each iteration
  - \( \frac{n}{10^{\text{len}(s)}} = 1 \) (i.e. divide by 10 until there is 1 element left to add)
  - \text{len}(s) = \log(n)

- \( \Theta(\log n) \) – base doesn’t matter

TRICKY COMPLEXITY

Linear \( \Theta(\text{len}(s)) \)
But what in terms of input n?
LOGARITHMIC COMPLEXITY

- Complexity grows as log of size of one of its inputs
- Example algorithm: binary search of a list
- Example we’ll see in a few slides: one bisection search implementation
# LIST AND DICTIONARIES

- Must be **careful** when using built-in functions!

<table>
<thead>
<tr>
<th>Lists – n is ( \text{len}(L) )</th>
<th>Dictionaries – n is ( \text{len}(d) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>index ( \Theta(1) )</td>
<td>index ( \Theta(1) )</td>
</tr>
<tr>
<td>store ( \Theta(1) )</td>
<td>store ( \Theta(1) )</td>
</tr>
<tr>
<td>length ( \Theta(1) )</td>
<td>length ( \Theta(1) )</td>
</tr>
<tr>
<td>append ( \Theta(1) )</td>
<td>delete ( \Theta(1) )</td>
</tr>
<tr>
<td>( \text{==} ) ( \Theta(n) )</td>
<td>( \text{.keys} ) ( \Theta(n) )</td>
</tr>
<tr>
<td>remove ( \Theta(n) )</td>
<td>( \text{.values} ) ( \Theta(n) )</td>
</tr>
<tr>
<td>copy ( \Theta(n) )</td>
<td>iteration ( \Theta(n) )</td>
</tr>
<tr>
<td>reverse ( \Theta(n) )</td>
<td></td>
</tr>
<tr>
<td>iteration ( \Theta(n) )</td>
<td></td>
</tr>
<tr>
<td>in list ( \Theta(n) )</td>
<td></td>
</tr>
</tbody>
</table>
SEARCHING ALGORITHMS
SEARCHING ALGORITHMS

- Linear search
  - **Brute force** search
  - List does not have to be sorted

- Bisection search
  - List **MUST be sorted** to give correct answer
  - Will see two different implementations of the algorithm
LINEAR SEARCH
ON UNSORTED LIST

```python
def linear_search(L, e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

- Must look through all elements to decide it’s not there
- $\Theta(len(L))$ for the loop * $\Theta(1)$ to test if $e == L[i]$
- Overall complexity is $\Theta(n)$ where $n$ is $len(L)$
- $\Theta(len(L))$

The loop goes through $len(L)$: $\Theta(len(L))$

Everything else is constant. $\Theta(1)$
LINEAR SEARCH
ON UNSORTED LIST

```python
def linear_search(L, e):
    for i in range(len(L)):
        if e == L[i]:
            return True
    return False
```

- Must look through all elements to decide it’s not there
- \( \Theta(len(L)) \) for the loop * \( \Theta(1) \) to test if \( e == L[i] \)
- Overall complexity is \( \Theta(n) \) where \( n \) is \( len(L) \)
- \( \Theta(len(L)) \)

Speed up a little by returning True here, but speed up doesn’t impact worst case.
LINEAR SEARCH ON SORTED LIST

```python
def search(L, e):
    for i in L:
        if i == e:
            return True
        if i > e:
            return False
    return False
```

- Must only look until reach a number greater than e
- $\Theta(\text{len}(L))$ for the loop * $\Theta(1)$ to test if $i == e$ or $i > e$
- Overall complexity is $\Theta(\text{len}(L))$
  $\Theta(n)$ where $n$ is $\text{len}(L)$

The loop goes through $\text{len}(L)$:
$\Theta(\text{len}(L))$

Everything else is constant.
$\Theta(1)$
BISECTION SEARCH FOR AN ELEMENT IN A **SORTED** LIST

1) Pick an index, $i$, that divides list in half
2) Ask if $L[i] == e$
3) If not, ask if $L[i]$ is larger or smaller than $e$
4) Depending on answer, search left or right half of $L$ for $e$

- A new version of **divide-and-conquer: recursion**!
- Break into smaller versions of problem (smaller list), plus simple operations
- Answer to smaller version is answer to original version
Bisection Search Complexity Analysis

- Finish looking through list when $1 = n/2^i$
- So... relationship between original length of list and how many times we divide the list: $i = \log n$
- Complexity is $\Theta(\log n)$ where $n$ is $\text{len}(L)$
BIG IDEA

Two different implementations have two different $\Theta$ values.
def bisect_search1(L, e):
    if L == []:
        return False
    elif len(L) == 1:
        return L[0] == e
    else:
        half = len(L)//2
        if L[half] > e:
            return bisect_search1(L[:half], e)
        else:
            return bisect_search1(L[half:], e)
COMPLEXITY OF bisect_search1 (where n is len(L))

- $\Theta(\log n)$ bisection search calls
  - Each recursive call cuts range to search in half
  - Worst case to reach range of size 1 from n is when $n/2^k = 1$ or when $k = \log n$
  - We do this to get an expression relating k to n
- $\Theta(n)$ for each bisection search call to copy list
  - Cost to set up recursive call at each level of recursion
- $\Theta(\log n) \times \Theta(n) = \Theta(n \log n)$ where $n = \text{len}(L)$
  ^ this is the answer in this class
- If careful, notice list is also halved on each recursive call
  - Infinite series (don’t worry about this in this class)
  - $\Theta(n)$ is a tighter bound because copying list dominates $\log n$
BISECTION SEARCH ALTERNATE IMPLEMENTATION

- Reduce size of problem by factor of 2 each step
- Keep track of low and high indices to search list
- Avoid copying list
- Complexity of recursion is $\Theta(\log n)$ where $n$ is $\text{len}(L)$
def bisect_search2(L, e):
    def bisect_search_helper(L, e, low, high):
        if high == low:
            return L[low] == e
        mid = (low + high) // 2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid:  # nothing left to search
                return False
            else:
                return bisect_search_helper(L, e, low, mid - 1)
        else:
            return bisect_search_helper(L, e, mid + 1, high)
    if len(L) == 0:
        return False
    else:
        return bisect_search_helper(L, e, 0, len(L) - 1)
COMPLEXITY OF bisect_search2 and helper (where n is len(L))

- $\Theta(\log n)$ bisection search calls
  - Each recursive call cuts range to search in half
  - Worst case to reach range of size 1 from n is when $n/2^k = 1$ or when $k = \log n$
  - We do this to get an expression relating $k$ to $n$

- Pass list and indices as parameters
  - List never copied, just re-passed
  - $\Theta(1)$ on each recursive call

- $\Theta(\log n) \times \Theta(1) = \Theta(\log n)$ where $n$ is len(L)
WHEN TO SORT FIRST AND THEN SEARCH?
SEARCHING A SORTED LIST
-- n is \text{len}(L)

- Using \textbf{linear search}, search for an element is \( \Theta(n) \)
- Using \textbf{binary search}, can search for an element in \( \Theta(\log n) \)
  - Assumes the list is sorted!
- When does it make sense to \textit{sort first then search}?

\begin{align*}
\text{Time to sort} & \quad \text{Time for binary search} \quad \text{Time for linear search} \\
\text{SORT} + \Theta(\log n) & < \Theta(n) \\
\text{implies that} \quad \text{SORT} & < \Theta(n) - \Theta(\log n)
\end{align*}

- When is sorting is less than \( \Theta(n) \)?
  \( \rightarrow \) Never true because you’d at least have to look at each element!
AMORTIZED COST
-- n is len(L)

- Why bother sorting first?
- **Sort a list once** then do **many searches**

- **AMORTIZE cost** of the sort over many searches

\[ \text{SORT} + K \cdot \Theta(\log n) < K \cdot \Theta(n) \]

implies that for large \( K \), **SORT time becomes irrelevant**
COMPLEXITY CLASSES SUMMARY

- Compare efficiency of algorithms
- Lower order of growth
- Using \( \Theta \) for an upper and lower ("tight") bound

Given a function \( f \):
- Only look at **items in terms of the input**
- Look at **loops**
  - Are they in terms of the input to \( f \)?
  - Are there nested loops?
- Look at **recursive calls**
  - How deep does the function call stack go?
- Look at **built-in functions**
  - Any of them depend on the input?