Complexity classes

- *O(1)* denotes constant running time
- *O(log n)* denotes logarithmic running time
- *O(n)* denotes linear running time
- *O(n log n)* denotes log-linear running time
- *O(n^c)* denotes polynomial running time (c is a constant)
- *O(c^n)* denotes exponential running time (c is a constant being raised to a power based on size of input)

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

Logarithmic complexity

- Complexity grows as log of size of one of its inputs
- Example:
  - Bisection search
  - Binary search of a list

Logarithmic complexity

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    result = ''
    while i > 0:
        result = digits[i%10] + result
        i = i/10
    return result
```

Logarithmic complexity

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    result = ''
    while i > 0:
        result = digits[i%10]
                 + result
        i = i/10
    return result
```

- Only have to look at loop as no function calls
- Within while loop constant number of steps
- How many times through loop?
  - How many times can one divide i by 10?
  - *O(log(i))*

Linear complexity

- Searching a list in order to see if an element is present
- Add characters of a string, assumed to be composed of decimal digits

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

- *O(len(s))*

Linear complexity

- Complexity can depend on number of recursive calls

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n*fact(n-1)
```

- Number of recursive calls?
  - Fact(n), then fact(n-1), etc. until get to fact(1)
  - Complexity of each call is constant
  - *O(n)*

Log-linear complexity

- Many practical algorithms are log-linear
- Very commonly used log-linear algorithm is merge sort
- Will return to this

Polynomial complexity

- 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

```
def isSubset(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
```

Quadratic complexity

```
def isSubset(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 len(L1) times
- Each iteration will execute inner loop up to len(L2) times
- *O(len(L1)*len(L2))*
- Worst case when L1 and L2 same length, none of elements of L1 in L2
- $O(len(L1)^2)$

Quadratic complexity

Find intersection of two lists, return a list with each element appearing only once

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

Quadratic complexity

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

- First nested loop takes *len(L1)*len(L2)* steps
- Second loop takes at most *len(L1)* steps
- Latter term overwhelmed by former term
- *O(len(L1)*len(L2))*

Exponential complexity

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

Exponential complexity

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]] #list of empty list
    smaller = genSubsets(L[:-1])
    # get all subsets without last element
    extra = L[-1:]
    # create a list of just last element
    new = []
    for small in smaller:
        new.append(small+extra)
    # for all smaller solutions, add one with last
  element
    return smaller+new
    # combine those with last element and those
  without
```

Exponential complexity

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

- Assuming append is constant time
- Time includes time to solve smaller problem, plus time needed to make a copy of all elements in smaller problem

Exponential complexity

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

- But important to think about size of smaller
- Know that for a set of size k there are 2^k cases
- So to solve need $2^{n-1} + 2^{n-2} + \ldots + 2^0$ steps
- Math tells us this is *$O(2^n)$*

Complexity classes

- *O(1)* denotes constant running time
- *O(log n)* denotes logarithmic running time
- *O(n)* denotes linear running time
- *O(n log n)* denotes log-linear running time
- *O(n^c)* denotes polynomial running time (c is a constant)
- *O(c^n)* denotes exponential running time (c is a constant being raised to a power based on size of input)