Running time

Which algorithm is faster depends on what the input is.
- If we know what inputs we'll have, just time them.
- Usually it's time on the big/hard inputs that matters.

We consider efficiency
- in the worst case
- for large inputs

Two algorithm run times are **asymptotically equal** if *for big inputs* which algorithm is faster *depends on the relative speed of the computers.*

Example:

Algorithm A has a worst-case running time on n-bit inputs of $n^3 - 4n^2$ steps.

Algorithm B has a worst-case running time on n-bit inputs of $10n^3 + 15$ steps.

If A and B are run on equal-speed machines, A is faster.

If B is run on a machine that is 100 times faster, B is faster (for large inputs).

Two algorithm run times are **asymptotically equal** if *for big inputs* which algorithm is faster *depends on the relative speed of the computers.*

$\theta(g(n))$ is the set of all functions asymptotically equal to g.

$f(n) \in \theta(g(n))$ means:

There exist positive constants k_1, k_2, m such that

$$k_1 f(n) < g(n) < k_2 f(n) \quad \text{for all } n > m$$

Primes

What's the running time?

```
def isPrime(n):
    i = 2
    while i < n:
        if n % i == 0:
            return False
        i += 1
    return True
```

Primes

What's the running time?

```
def sumPrimes(n):
    i = 2
    total = 0
    while i < n:
        if isPrime(i):
            total += i
    return total
```

Primes

What's the running time?

```
def isPrime(n):
    i = 2
    while i < sqrt(n):
        if n % i == 0:
            return False
        i += 1
    return True


def sumPrimes(n):
    i = 2
    total = 0
    while i<n:
        if isPrime(i):
            total += i
    return total
```

Searching

Given a sorted list and a value v, does the list include v?

Option 1:

```
def search(ls, v):
    for i in ls:
        if i == v:
            return True
    return False
```

Searching

Given a sorted list and a value v, does the list include v?

Option 2:

```
def search_help(ls, v, start, end):
    mid = (start + end) // 2
    if ls[mid] == v:
        return True
    elif start >= end:
        return False
    elif ls[mid] > v:
        return search_help(ls, v, start, mid-1)
    else:
        return search_help(ls, v, mid+1, end)

def search(ls, v):
    return search_help(ls, v, 0, len(ls)-1)
```

Sorting

We want to put a list in order.

Ideas?

Bubble Sort

```
def bubblesort(aList):
   for i in range(len(aList)):
      for k in range(len(aList)-1):
         if aList[k] > aList[k+1]:
            aList[k], aList[k+1] = aList[k+1], aList[k]
   return aList
```

Bubble Sort

```
def bubblesort(aList):
   for i in range(len(aList)):
      for k in range(len(aList)-1-i):
         if aList[k] > aList[k+1]:
            aList[k], aList[k+1] = aList[k+1], aList[k]
   return aList
```

Bubble Sort

```
def bubblesort(aList):
   for i in range(len(aList)):
      finished = True
      for k in range(len(aList)-1-i):
         if aList[k] > aList[k+1]:
            finished = False
            aList[k], aList[k+1] = aList[k+1], aList[k]
      if finished:
         break
   return aList
```

Insertion Sort

```
def insertionsort(aList):
   for i in range( 1, len( aList ) ):
      tmp = aList[i]
      k = i
      while k > 0 and tmp < aList[k - 1]:
         aList[k] = aList[k - 1]
         k -= 1
      aList[k] = tmp
   return aList
```

Merge Sort

```
def merge(a, b):
   c = []
   i, j = 0, 0
   while i + j < len(a) + len(b):
      if i >= len(a) or (j < len(b) and b[j] <= a[i]):
         c.append(b[j])
         j += 1
      elif j >= len(b) or a[i] <= b[j]:
         c.append(a[i])
         i += 1
   return c

def mergesort(aList):
   if len(aList) <= 1:
      return aList
   else:
      mid = len(aList) // 2
      return merge(mergesort(aList[:mid]),mergesort(aList[mid:]))
```