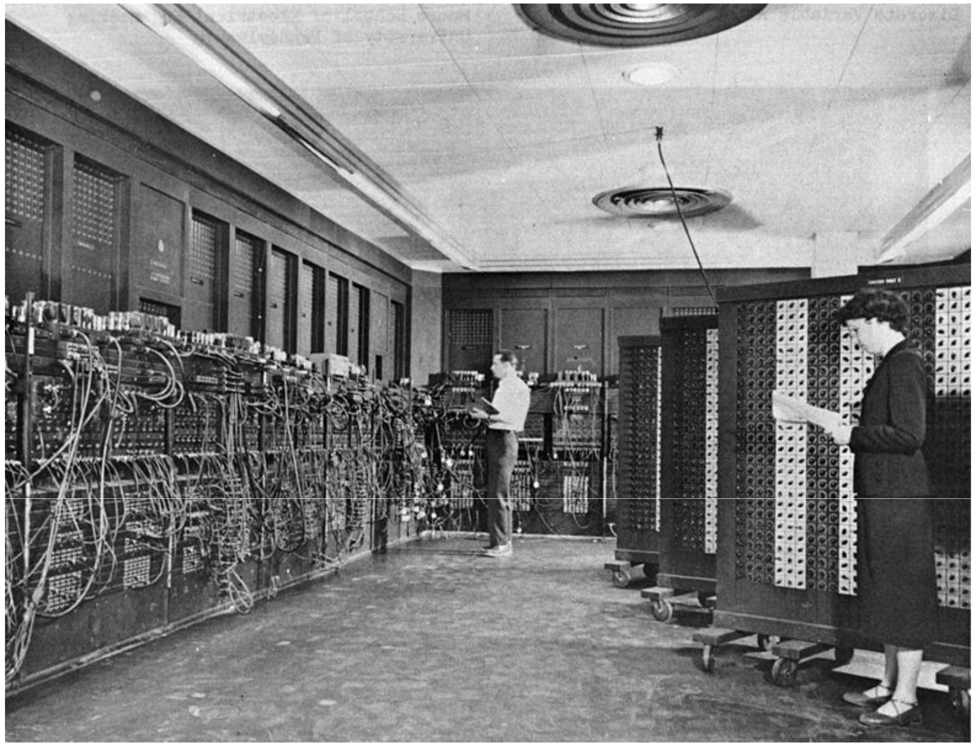
UNIVERSITAT DE BARCELONA

Grau d'Enginyeria Informàtica
Facultat de Matemàtiques



Algorísmica

Introducció als algorismes III

Jordi Vitrià

Col·leccions de dades i Python

Exemples de **col·leccions**:

- Paraules d’un text.
- Estudiants d’un curs.
- Dades d’un experiment.
- Clients d’un negoci.
- Els gràfics que es poden dibuixar en una finestra.

Python ens dona suport per a la manipulació d’aquest tipus de dades.

Col·leccions de dades i Python

```
# average4.py
def main():
    sum = 0.0
    count = 0
    xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = eval(xStr)
        sum = sum + x
        count = count + 1
        xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    print "\nThe average of the numbers is", sum / count

main()
```

Avaluem una expressió representada a un string

Suposem que també volem calcular la mediana i la desviació estàndard....

$$s = \sqrt{\frac{\sum(\bar{x} - x_i)^2}{n-1}}$$

Necessitem guardar tots els nombres que han entrat.

Col·leccions de dades i Python

El que necessitem és emmagatzemar una col·lecció de coses (a priori no sabem quantes) en un “objecte”.

De fet, aquest tipus d’”objecte” ja l’hem fet servir, i es diu **llista**:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> string.split("This is an ex-parrot!")
['This', 'is', 'an', 'ex-parrot!']
```

Una llista és una seqüència ordenada de coses.

$$S = s_0, s_1, s_2, s_3, \ldots, s_{n-1}$$

Els elements estan indexats per subíndexos

Col·leccions de dades i Python

De fet les llistes i els *strings* són conceptualment molt semblants, i podem aplicar-hi operadors semblants:

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[ ]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration

La diferència és el que contenen. Les llistes **poden contenir qualsevol tipus de dades**, incloent "classes" definides pel programador. Les llistes són **mutables**, és a dir, es poden canviar sobre la mateixa estructura (els *strings* no!).

```
>>> myList = [34, 26, 15, 10]
>>> myList[2]
15
>>> myList[2] = 0
>>> myList
[34, 26, 0, 10]
>>> myString = "Hello World"
>>> myString[2]
'l'
>>> myString[2] = 'z'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Col·leccions de dades i Python

Les llistes en Python són **dinàmiques**, poden créixer i decréixer durant l’execució del programa.

Les llistes en Python són **inhomogènies**, poden contenir tipus diferents de dades.

En resum, les llistes són seqüències mutables d’objectes arbitraris.

Es creen així:

```
odds = [1, 3, 5, 7, 9]
food = ["spam", "eggs", "back bacon"]
silly = [1, "spam", 4, "U"]
empty = []
```

Col·leccions de dades i Python

Podem crear llistes d’objectes idèntics així:

```
zeroes = [0] * 50
```

O afegir-hi/borra-hi coses:

```
nums = []
x = input(’Enter a number: ’)
while x >= 0:
    nums.append(x)
    x = input("Enter a number: ")
```

```
>>> myList
[34, 26, 0, 10]
>>> del myList[1]
>>> myList
[34, 0, 10]
>>> del myList[1:3]
>>> myList
[34]
```

Col·leccions de dades i Python

Method	Meaning
< `list` >.append(x)	Add element x to end of list.
< `list` >.sort()	Sort the list. A comparison function may be passed as parameter.
< `list` >.reverse()	Reverses the list.
< `list` >.index(x)	Returns index of first occurrence of x.
< `list` >.insert(i,x)	Insert x into list at index i. (Same as `list[i:i] = [x]`)
< `list` >.count(x)	Returns the number of occurrences of x in list.
< `list` >.remove(x)	Deletes the first occurrence of x in list.
< `list` >.pop(i)	Deletes the ith element of the list and returns its value.
x in <*list* >	Checks to see if x is in the list (returns a Boolean).

```
>>> z=[0] * 10
>>> z
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> z.insert(1,1)
>>> z
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> 1 in z
True
>>> y = z.pop(1)
>>> y
1
>>> z
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Col·leccions de dades i Python

Amb el que sabem podem reescriure el codi del càlcul estadístic.
Primer, recollim les dades de l’usuari:

```
def getNumbers():
    nums = []     # start with an empty list

    # sentinel loop to get numbers
    xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = eval(xStr)
        nums.append(x)   # add this value to the list
        xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    return nums
```

Col·leccions de dades i Python

Després calculem la mitja:

```
def mean(nums):
    sum = 0.0
    for num in nums:
        sum = sum + num
    return sum / len(nums)
```

I el programa original queda:

```
def main():
    data = getNumbers()
    print 'The mean is', mean(data)
```

Un cop tenim la mitja podem calcular la desviació :

```
def stdDev(nums, xbar):
    sumDevSq = 0.0
    for num in nums:
        dev = xbar - num
        sumDevSq = sumDevSq + dev * dev
    return sqrt(sumDevSq/(len(nums)-1))
```

$$s = \sqrt{\frac{\sum(\bar{x} - x_i)^2}{n-1}}$$

Col·leccions de dades i Python

La mediana és una mica més complicada.

```
sort the numbers into ascending order
if the size of  data is odd:
    median = the middle value
else:
    median = the average of the two middle values
return median
```

```
def median(nums):
    nums.sort()
    size = len(nums)
    midPos = size / 2
    if size % 2 == 0:
        median = (nums[midPos] + nums[midPos-1]) / 2.0
    else:
        median = nums[midPos]
    return median
```

Col·leccions de dades i Python

```
def main():
    print 'This program computes mean, median and standard deviation.'

    data = getNumbers()
    xbar = mean(data)
    std = stdDev(data, xbar)
    med = median(data)

    print '\nThe mean is', xbar
    print 'The standard deviation is', std
    print 'The median is', med
```

Col·leccions de dades i Python

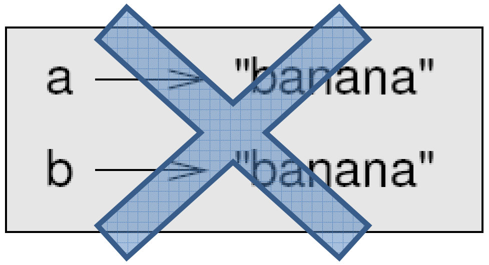
Si executem:

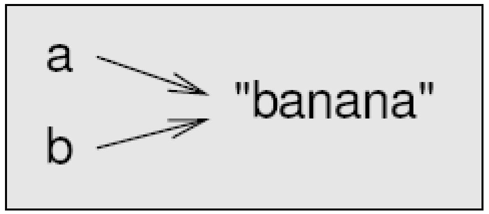
```
a = “banana”
b = “banana”
```

a i b “apunten” a un *string* amb el mateix valor, però és el mateix *string*?

Cada objecte té un identificador únic, que podem obtenir amb la funció id:

```
>>> id(a)
135044008
>>> id(b)
135044008
```



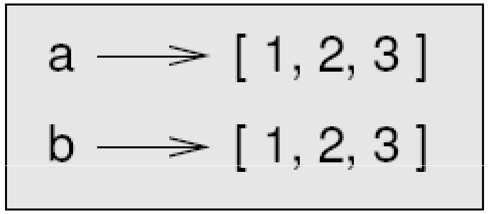


Per tant, en aquest cas Python ha creat una estructura “banana” i les dues variables en fan referència.

Col·leccions de dades i Python

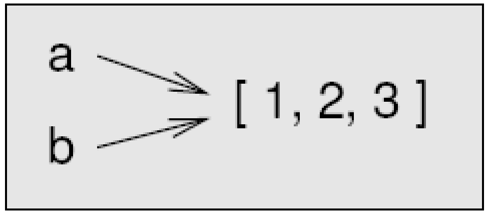
Les **llistes** funcionen diferent (a i b tenen el mateix valor però no es refereixen al mateix objecte):

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```



Com que les variables es refereixen a objectes, si fem referir una variable a una altra tenim:

```
>>> a = [1, 2, 3]
>>> b = a
```



Col·leccions de dades i Python

Com que la llista té dos noms, direm que té un alias:

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

Això és perillós per objectes mutables!!! Pels immutables no hi ha problema (*strings*).

El **clonatge** és una tècnica per la que fem una còpia de l'objecte en si, no de la referència. Pel cas de les llistes ho podem fer així:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print b
[1, 2, 3]
```

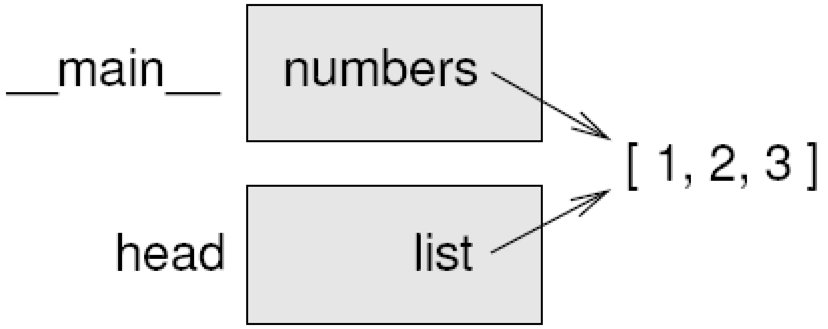
```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

Col·leccions de dades i Python

Si passem una llista com argument d'una funció, **passem una referència, no una còpia**. Considerem aquesta funció:

```
def head(list):
  return list[0]
```

```
>>> numbers = [1, 2, 3]
>>> head(numbers)
1
```



Col·leccions de dades i Python

Considerem aquesta altra funció:

```
def deleteHead(list):
  del list[0]
```

```
>>> numbers = [1, 2, 3]
>>> deleteHead(numbers)
>>> print numbers
[2, 3]
```

Si retornem una llista també retornem una referència:

```
def tail(list):
  return list[1:]
```

```
>>> numbers = [1, 2, 3]
>>> rest = tail(numbers)
>>> print rest
[2, 3]
```

Com que la llista **s'ha creat amb : és una nova llista**. Qualsevol modificació de rest no té efectes a `numbers`.

Col·leccions de dades i Python

```
>>> numbers = [1, 2, 3]
>>> def test(l):
...     return l.reverse()
...
>>> test(numbers)
>>> numbers
[3, 2, 1]
>>>
```

Col·leccions de dades i Python

Una **llista nidada** és una llista que apareix com a element d’una altra llista.

```
>>> list = ["hello", 2.0, 5, [10, 20]]
```

Per obtenir un element d’una llista nidada ho podem fer de dues maneres:

```
>>> elt = list[3]
>>> elt[0]
10
```

```
>>> list[3][1]
20
```

Les llistes nidades es fan servir per representar matrius

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
>>> matrix[1]
[4, 5, 6]
```

```
>>> matrix[1][1]
5
```

Col·leccions de dades i Python

Python ens proporciona un altre tipus de col·lecció no gaire estàndard, però molt útil: els diccionaris.

La raó de la seva existència és que no sempre serà possible accedir a una dada pel seu índex, sinó per exemple, pel seu valor (p.e. pel DNI d'un conjunt d'empleats).

És a dir,volem accedir a un valor per una clau.

Una col·lecció que ens permet això es diu un "*mapping*" (altres llenguatges ho anomenen *taules hash* o *vectors associatius*).

Python les crea així:

```
>>> passwd = {"guido":"superprogrammer", "turing":"genius", "bill":"monopoly"}
```

Col·leccions de dades i Python

I ens permet accedir-hi així:

```
>>> passwd["guido"]
'superprogrammer'
>>> passwd["bill"]
'monopoly'
```

Els diccionaris són mutables:

```
>>> passwd["bill"] = "bluescreen"
>>> passwd
{'turing': 'genius', 'bill': 'bluescreen', 'guido': 'superprogrammer'}
```

(els diccionaris no tenen ordre, i Python els imprimeix amb un ordre propi, no el que hem entrat!)

Col·leccions de dades i Python

Podem entrar-hi dades:

```
>>> passwd['newuser'] = 'ImANewbie'
>>> passwd
{'turing': 'genius', 'bill': 'bluescreen', \
 'newuser': 'ImANewbie', 'guido': 'superprogrammer'}
```

..des d'un fitxer:

```
>>> f = open('passwords.txt', 'r')
>>> for line in f.readlines():
...     print line,
...
jordi vitria
bill clinton
>>>
```

Llegeix strings

```
passwd = {}
for line in open('passwords','r').readlines():
    user, pass = string.split(line)
    passwd[user] = pass
```

Col·leccions de dades i Python

Method	Meaning
`<dict>.has_key(<key>)`	Returns true if dictionary contains the specified key, false if it doesn't.
`<dict>.keys()`	Returns a list of the keys.
`<dict>.values()`	Returns a list of the values.
`<dict>.items()`	Returns a list of tuples `(key,value)` representing the key-value pairs.
`del <dict>[<key>]`	Delete the specified entry.
`<dict>.clear()`	Delete all entries.

```
>>> passwd.keys()
['turing', 'bill', 'newuser', 'guido']
>>> passwd.values()
['genius', 'bluescreen', 'ImANewbie', 'superprogrammer']
>>> passwd.items()
[('turing', 'genius'), ('bill', 'bluescreen'), ('newuser', 'ImANewbie'), \
 ('guido', 'superprogrammer')]
>>> passwd.has_key('bill')
1
>>> passwd.has_key('fred')
0
>>> passwd.clear()
>>> passwd
{}
```

Col·leccions de dades i Python

Hi ha una altra classe de col·lecció a Python que és semblant a la llista, però que és immutable.: la tupla.

```
>>> tuple = 'a', 'b', 'c', 'd', 'e'
```

Que també es pot (i es sol) escriure com:

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
```

Si només té un element hem de posar-hi una coma, sinó crea un string!

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Col·leccions de dades i Python

Les operacions són les mateixes que per les llistes (tenint en compte que són **immutables**!):

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
>>> tuple[0]
'a'

>>> tuple[1:3]
('b', 'c')
```

```
>>> tuple[0] = 'A'
TypeError: object doesn't support item assignment
```

```
>>> tuple = ('A',) + tuple[1:]
>>> tuple
('A', 'b', 'c', 'd', 'e')
```

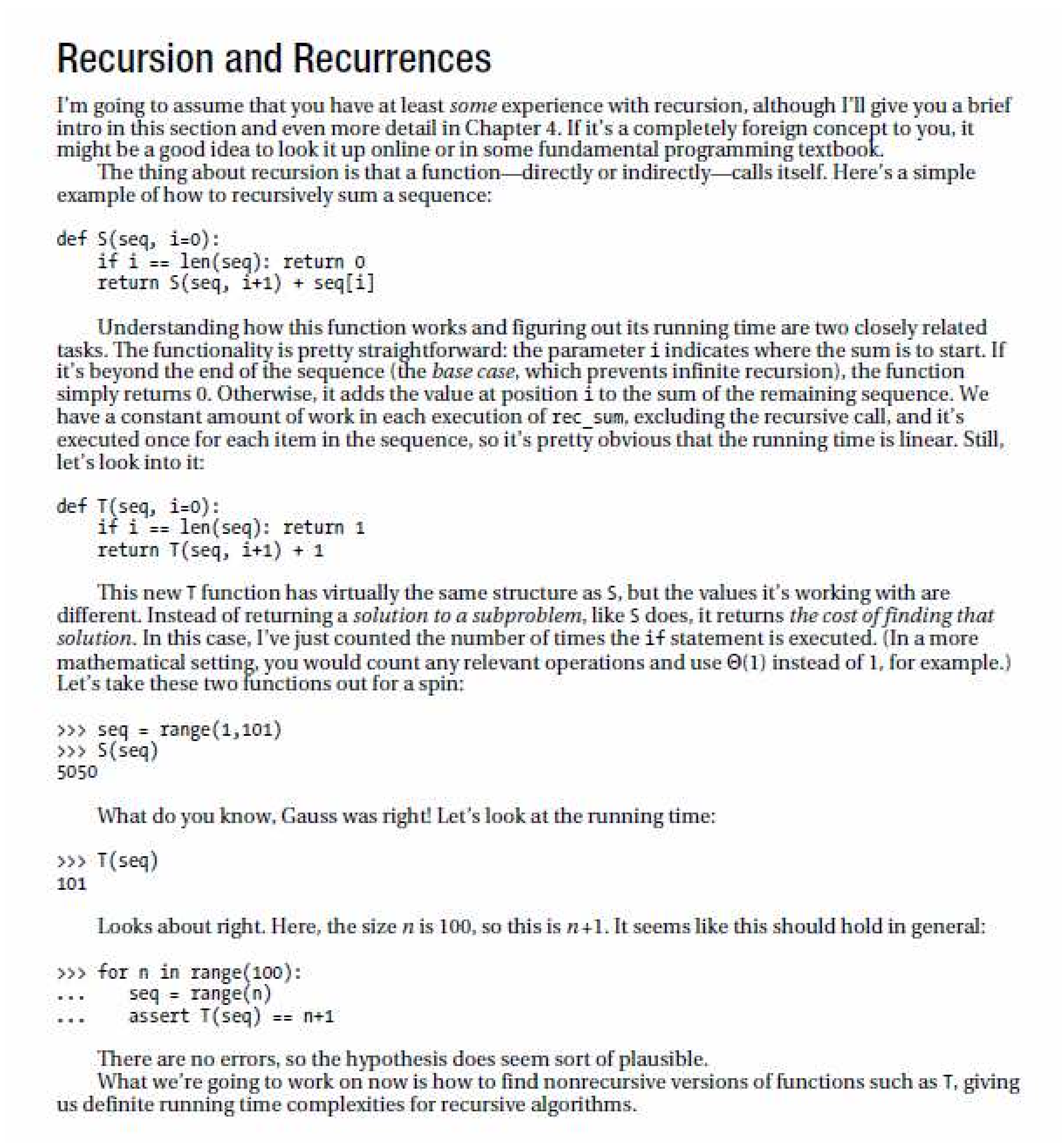
Col·leccions de dades i Python

Exemple: com fer l’estadística de les paraules que apareixen en un document.

```
# wordfreq.py
import string

def compareItems((w1,c1), (w2,c2)):
    if c1 > c2:
        return - 1
    elif c1 == c2:
        return cmp(w1, w2)
    else:
        return 1

def main():
    print "This program analyzes word frequency in a file"
    print "and prints a report on the n most frequent words.\n"

    # get the sequence of words from the file
    fname = raw_input("File to analyze: ")
    text = open(fname,’r’).read()
    text = string.lower(text)
    for ch in ’!"#$%&()*+,-./:;<=>?@[\\]^_‘{|}~’:
        text = string.replace(text, ch, ’ ’)
    words = string.split(text)

    # construct a dictionary of word counts
```

Col·leccions de dades i Python

```
    counts = {}
    for w in words:
        try:
            counts[w] = counts[w] + 1
        except KeyError:
            counts[w] = 1

    # output analysis of n most frequent words.
    n = input("Output analysis of how many words? ")
    items = counts.items()
    items.sort(compareItems)
    for i in range(n):
        print "%-10s%5d" % items[i]

if __name__ == '__main__': main()
```



Recursion and Recurrences

I'm going to assume that you have at least *some* experience with recursion, although I'll give you a brief intro in this section and even more detail in Chapter 4. If it's a completely foreign concept to you, it might be a good idea to look it up online or in some fundamental programming textbook.

The thing about recursion is that a function—directly or indirectly—calls itself. Here's a simple example of how to recursively sum a sequence:

```
def S(seq, i=0):
    if i == len(seq): return 0
    return S(seq, i+1) + seq[i]
```

Understanding how this function works and figuring out its running time are two closely related tasks. The functionality is pretty straightforward: the parameter `i` indicates where the sum is to start. If it's beyond the end of the sequence (the *base case*, which prevents infinite recursion), the function simply returns 0. Otherwise, it adds the value at position `i` to the sum of the remaining sequence. We have a constant amount of work in each execution of `rec_sum`, excluding the recursive call, and it's executed once for each item in the sequence, so it's pretty obvious that the running time is linear. Still, let's look into it:

```
def T(seq, i=0):
    if i == len(seq): return 1
    return T(seq, i+1) + 1
```

This new `T` function has virtually the same structure as `S`, but the values it's working with are different. Instead of returning a *solution to a subproblem*, like `S` does, it returns *the cost of finding that solution*. In this case, I've just counted the number of times the `if` statement is executed. (In a more mathematical setting, you would count any relevant operations and use $\Theta(1)$ instead of 1, for example.) Let's take these two functions out for a spin:

```
>>> seq = range(1,101)
>>> S(seq)
5050
```

What do you know, Gauss was right! Let's look at the running time:

```
>>> T(seq)
101
```

Looks about right. Here, the size n is 100, so this is $n+1$. It seems like this should hold in general:

```
>>> for n in range(100):
...     seq = range(n)
...     assert T(seq) == n+1
```

There are no errors, so the hypothesis does seem sort of plausible.

What we're going to work on now is how to find nonrecursive versions of functions such as `T`, giving us definite running time complexities for recursive algorithms.

Doing It by Hand

To describe the running time of recursive algorithms mathematically, we use recursive equations, called *recurrence relations*. If our recursive algorithm is like `S` in the previous section, then the recurrence relation is defined somewhat like `T`. Because we're working toward an asymptotic answer, we don't care about the constant parts, and we implicitly assume that $T(k) = \Theta(1)$, for some constant k. That means that we can ignore the base cases when setting up our equation (unless they *don't* take a constant amount of time), and for `S`, our T can be defined as follows:

$$T(n) = T(n-1) + 1$$

This means that the time it takes to compute `S(seq, i)` (which is $T(n)$) is equal to the time required for the recursive call `S(seq, i+1)` (which is $T(n-1)$) plus the time required for the access `seq[i]` (which is constant, or $\Theta(1)$). Put another way, we can reduce the problem to a smaller version of itself (from size n to $n-1$) in constant time and then solve the smaller subproblem. The total time is the sum of these two operations.

Note As you can see, I use 1 rather than $\Theta(1)$ for the extra work (that is, time) outside the recursion. I could use the theta as well; as long as I describe the result asymptotically, it won't matter much. In this case, using $\Theta(1)$ might be risky, because I'll be building up a sum (1 + 1 + 1 …), and it would be easy to mistakenly simplify this sum to a constant if it contained asymptotic notation (that is, $\Theta(1) + \Theta(1) + \Theta(1)$ …).

Now, how do we *solve* an equation like this? The clue lies in our implementation of T as an executable function. Instead of having Python run it, we can simulate the recursion ourselves. The key to this whole approach is the following equation:

$$\begin{aligned} T(n) &= \boxed{T(n-1)} + 1 \\ &= \boxed{T(n-2)+1} + 1 \\ &= T(n-2) + 2 \end{aligned}$$

The two subformulas I've put in boxes are identical, which is sort of the point. My rationale for claiming that the two boxes are the same lies in our original recurrence, for if…

$$T(n) = T(n-1) + 1$$

… then:

$$\boxed{T(n-1)} = \boxed{T(n-2)+1}$$

I've simply replaced n with $n-1$ in the original equation (of course, $T((n-1)-1) = T(n-2)$), and *voilà*, we see that the boxes are equal. What we've done here is to use the definition of T with a smaller parameter, which is, essentially, what happens when a recursive call is evaluated. So, expanding the recursive call from $T(n-1)$ (the first box) to $T(n-2) + 1$ (the second box) is essentially simulating or "unraveling" one level of recursion. We still have the recursive call $T(n-2)$ to contend with, but we can deal with that in the same way!

$$\begin{aligned} T(n) &= T(n-1)+1 \\ &= \boxed{T(n-2)}+2 \\ &= \boxed{T(n-3)+1}+2 \\ &= T(n-3)+3 \end{aligned}$$

The fact that $T(n-2) = T(n-3) + 1$ (the two boxed expressions) again follows from the original recurrence relation. It's at this point we should see a pattern: each time we reduce the parameter by one, the sum of the work (or time) we've unraveled (outside the recursive call) goes *up* by one. If we unravel $T(n)$ recursively i steps, we get the following:

$$T(n) = T(n-i)+i$$

This is *exactly* the kind of expression we're looking for—one where the level of recursion is expressed as a variable i. Because all these unraveled expressions are equal (we've had equations every step of the way), we're free to set i to any value we want, as long as we don't go past the base case (for example, $T(1)$), where the original recurrence relation is no longer valid. What we do is go *right up to* the base case and try to make $T(n-i)$ into $T(1)$, because we know (or implicitly assume) that $T(1)$ is $\Theta(1)$, which would mean we had solved the entire thing. And we can easily do that by setting $i = n-1$:

$$\begin{aligned} T(n) &= T(n-(n-1))+(n-1) \\ &= T(1)+n-1 \\ &= \Theta(1)+n-1 \\ &= \Theta(n) \end{aligned}$$

We have now, with perhaps more effort than was warranted, found that S has a linear running time, as we suspected. In the next section, I'll show you how to use this method for a couple of recurrences that aren't quite as straightforward.

■ **Caution** This method, called the method of *repeated substitutions* (or sometimes the *iteration method*), is perfectly valid, if you're careful. However, it's quite easy to make an unwarranted assumption or two, *especially* in more complex recurrences. This means that you should probably treat the result as a *hypothesis* and then check your answer using the techniques described in the section "Guessing and Checking" later in this chapter.

A Few Important Examples

The general form of the recurrences you'll normally encounter is is $T(n) = a \cdot T(g(n)) + f(n)$, where a represents the number of recursive calls, $g(n)$ is the size of each subproblem to be solved recursively, and $f(n)$ is any extra work done in the function, in addition to the recursive calls.

Tip It's certainly possible to formulate recursive algorithms that don't fit this schema, for example if the subproblem sizes are different. Such cases won't be dealt with in this book, but some pointers for more information are given in the section "If You're Curious…," near the end of this chapter.

Table 3-1 summarizes some important recurrences (one or two recursive calls on problems of size n–1 or $n/2$, with either constant or linear additional work in each call). You've already seen recurrence number 1 in the previous section. In the following, I'll show you how to solve the last four using repeated substitutions, leaving the remaining three (2 to 4) for Exercises 3-7 to 3-9.

Table 3-1. Some Basic Recurrences with Solutions, as Well as Some Sample Applications

#	Recurrence	Solution	Example Applications
1	$T(n) = T(n-1) + 1$	$\Theta(n)$	Processing a sequence, for example, with reduce
2	$T(n) = T(n-1) + n$	$\Theta(n^2)$	Handshake problem
3	$T(n) = 2T(n-1) + 1$	$\Theta(2^n)$	Towers of Hanoi
4	$T(n) = 2T(n-1) + n$	$\Theta(2^n)$	
5	$T(n) = T(n/2) + 1$	$\Theta(\lg n)$	Binary search (see the black box sidebar on `bisect` in Chapter 6)
6	$T(n) = T(n/2) + n$	$\Theta(n)$	Randomized Select, average case (see Chapter 6)
7	$T(n) = 2T(n/2) + 1$	$\Theta(n)$	Tree traversal (see Chapter 5)
8	$T(n) = 2T(n/2) + n$	$\Theta(n \lg n)$	Sorting by *divide and conquer* (see Chapter 6)

Before we start working with the last four recurrences (which are all examples of *divide and conquer* recurrences, explained more in detail later in this chapter and in Chapter 6), you might want to refresh your memory with Figure 3-5. It summarizes the results I've discussed so far about binary trees; sneakily enough, I've already given you all the tools you need (as you'll see in the following).

Note I've already mentioned the assumption that the base case has constant time ($T(k) = t_0$, $k \leq n_0$, for some constants t_0 and n_0). In recurrences where the argument to T is n/b, for some constant b, we run up against another technicality: the argument really should be an integer. We could achieve that by rounding (using `floor` and `ceil` all over the place), but it's common to simply ignore this detail (really assuming that n is a power of b). To remedy the sloppiness, you should check your answers with the method described in "Guessing and Checking" later in this chapter.