Unit 6

Algorithms

Learning Outcomes

- Use and compare two algorithms for searching in lists.
- Use sorting to solve problems.
- Design and implement functions that use recursion.
- Understand and implement a simple sorting algorithm.
- Describe some problems that aren't computable.
- Use recursion to solve problems and implement algorithms.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Sections 4.9–4.11 in *How to Think Like a Computer Scientist*.

Topic 6.1 Searching

Searching is an important program in computing. "Searching" is the problem of looking up a particular value in a list or other data structure. You generally want to find the value (if it's there) and determine its position.

We will only worry about searching in lists here. There are many other data structures that can be used to store data; each one has its own searching algorithms that can be applied.

```
def search(lst, val):
    """
    Find the first occurrence of val in lst.  Return its
    index or -1 if not there.

    >>> search([0, 10, 20, 30, 40], 30)
    3
    >>> search([0, 10, 20, 30, 40], 25)
    -1
    """
    for i in range(len(lst)):
        if lst[i]==val:
            # we only care about the first match,
            # so if we've found one, return it.
            return i

    # if we get this far, there is no val in lst.
    return -1
```

Figure 6.1: Python implementation of linear search

Linear Search

For lists in general, you have to look through the whole list to determine if the value is present or not. The algorithm for this is simple: just search through the list from element 0 to the end: if you find the value you're looking for, stop; if you never do, return -1. (We will always use the "position" -1 to indicate "not found".)

This search algorithm is called *linear search* and a Python implementation can be found in Figure 6.1.

What's the running time of a linear search for a list with n items? At worst, it will have to scan through each of the n elements, checking each one. So, the running time is n.

This isn't too bad, but if you have to do a lot of lookups in a list, it will take a lot of time. We can do better if the list is arranged properly.

The list method `lst.index(val)` does a linear search to find the position of `val` in the list `lst`. If the value isn't there, it causes a `ValueError` instead of returning -1 like Figure 6.1. The "`in`" operator (`if val in lst:`) also uses linear search.

Binary Search

Think about how you look up numbers in a phone book.

If you are trying to find out who has a particular *number*, you have to look through the whole book like the linear search does. Starting with the first person in the phone book and scan all of the phone numbers until you find the phone number you're looking for, or get to the end of the book. So, it's possible to use the phone book to translate a phone number to the person's name that owns it, but it's very impractical.

On the other hand, what you usually do with a phone book is translate a persons *name* to a phone number. This is a lot easier because phone books are sorted by name, so you don't have to scan every entry; you can quickly find the person you're looking for.

So, if we have a Python list that's in order, we should be able to take advantage of this to search faster. We can write an algorithm that formalizes what you do with a phone book: use the fact that it's sorted to find the right general part of the book, then the right page, the right column, and the right name.

The algorithm that is used to search in sorted lists has a lot in common with the guessing game that was designed in Figure 1.3 and implemented in Figure 3.11. You can think of this game as searching for the value the user was thinking of in the list `[1, 2, 3, ..., 100]`.

The search algorithm will use the same strategy: keep track of the first and last possible position that the value you're looking for can be. This will start at the first and last items in the list, but can be narrowed quickly.

Then, look at the list item that's halfway between the first and last possible values. If it's too large, then you know that none of the values after it in the list are possibilities: they are *all* too large. Now you only have to look at the first half of the list, so the problem has immediately been chopped in half.

Similarly, if the value we check in the middle is too small, then we only have to look at values after it in the list. In the guessing game, we did the

exact same thing, except we had to ask the user to enter "less", "more", or "equal". Here, you can just check the value in the list.

This algorithm is called *binary search*. A Python implementation of binary search can be found in Figure 6.2.

Just like the original guessing game, this algorithm cuts the list we're searching in half with each iteration. So, like that algorithm, it has running time $\log n$.

Have a look back at Figure 3.17 for a comparison of n (linear search) and $\log n$ (binary search). As you can see, the binary search will be *much* faster for large lists.

But, to use binary search, you have to keep the list in sorted order. That means that you can't just use `list.append()` to insert something into the list anymore. New values have to be inserted into their proper location, so inserting takes a lot more work

Inserting into a sorted list takes up to n steps because Python has to shuffle the existing items in the list down to make room. You don't see this since you can just use `list.insert()`, but it does happen behind-the-scenes.

This means that keeping a sorted list and doing binary search is only worthwhile if you need to search a lot more than you insert. If you have some data that doesn't change very often, but need to find values regularly, it's more efficient to keep the list sorted because it makes searches so much faster.

Searching is covered in more detail in CMPT 225 and 307. These courses discuss other data structures and how they can be used to hold sets of data so searching, inserting, and deleting are all efficient. There are data structures that can do insert, search, and delete all with running time $\log n$, but they are complicated and aren't covered until CMPT 307.

Topic 6.2 Sorting

Sorting is another important problem in computing science. It's something that comes up very often, in a variety of situations. There are many problems that can be solved quite quickly by first sorting the values you need to work with: once the values are in order, many problems become a lot easier.

```
def binary_search(lst, val):
    """
    Find val in lst.  Return its index or -1 if not there.
    The list MUST be sorted for this to work.

    >>> binary_search([2, 4, 5, 6, 24, 100, 1001], 100)
    5
    >>> binary_search([2, 4, 5, 6, 24, 100, 1001], 10)
    -1
    >>> binary_search([2, 4, 5, 6, 24, 100, 1001], 2000)
    -1
    """

    # keep track of the first and last possible positions.
    first = 0
    last = len(lst)-1

    while first <= last:
        mid = (first+last)/2
        if lst[mid] == val:
            # found it
            return mid
        elif lst[mid] < val:
            # too small, only look at the right half
            first = mid+1
        else: # lst[mid] > val
            # too large, only look at the left half
            last = mid-1

    # if we get this far, there is no val in lst.
    return -1
```

Figure 6.2: Python implementation of binary search

```
word = raw_input("Enter the word: ")
counter = 0

letters = list(word) # converts to a list of characters.
                     # 'gene' becomes ['g','e','n','e']
letters.sort()       # now identical letters are adjacent
                     # above becomes ['e','e','g','n']

for i in range(len(word)-1):
    if letters[i]==letters[i+1]:
        counter = counter + 1

if counter>0:
    print "There are repeated letters"
else:
    print "There are no repeated letters"
```

Figure 6.3: Checking for repeated letters with sorting

In the previous topic, you saw that searching is much faster if the list is sorted first. Sorting takes longer than even a linear search, but if there are going to be many searches, it is worth the work.

A list in Python can be put in order by calling its `sort` method:

```
>>> mylist = [100, -23, 12, 8, 0]
>>> mylist.sort()
>>> print mylist
[-23, 0, 8, 12, 100]
```

Example: Repeated letters with sorting

As an example of a problem where sorting can greatly speed up a solution, recall the problem of finding repeated letters in a string. Figure 3.13 gives an algorithm with running time n^2 and Figure 3.14 is a Python implementation.

Now consider the program in Figure 6.3. It does the same thing as the program in Figure 3.14, but it will be significantly faster than the previous solution for long strings.

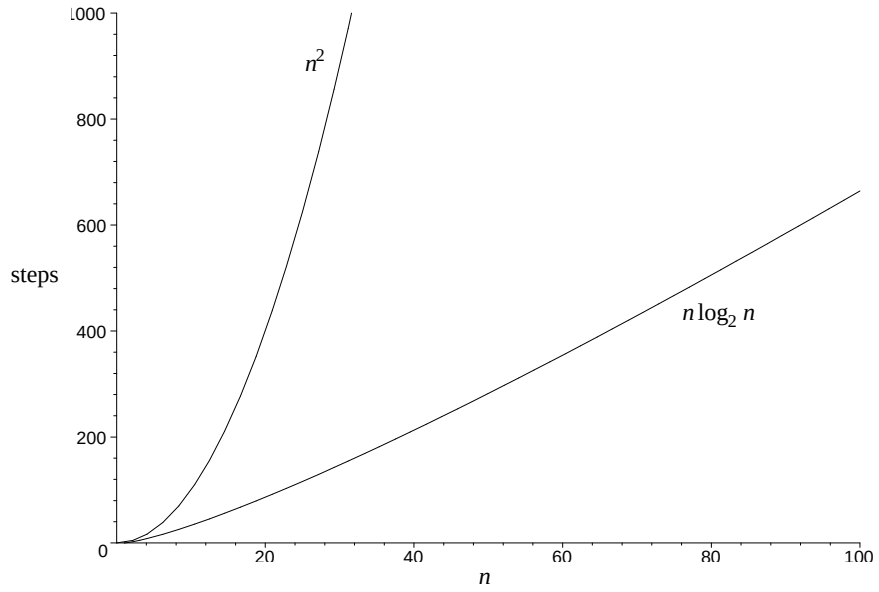

Figure 6.4: Graph of the functions n^2 and $n \log_2 n$

The idea is that the program first sorts the letters of the string. Then, any identical letters will be beside each other. So, to check for repeated letters, you only have to skim through the characters once, looking at characters `i` and `i+1` to see if they are the same. If none are, then there are no repeated letters.

The `sort` method has running time $n \log n$ on a list with n elements. The rest of the program just scans once through the list, so it takes n steps. The total running time for Figure 6.3 will be $n \log n + n$. Removing the lower-order terms, we get $n \log n$.

See Figure 6.4 for a comparison of n^2 and $n \log n$. As you can see, the new program that takes advantage of a fast sorting algorithm will be much faster as n grows.

How to sort

Usually, you can use the `sort` method that comes with a list in Python when you need to get items in order. But, sorting is important enough that you should have some idea of how it's done.

As noted above, the `sort` method of a list has running time $n \log n$. In general, it's not possible to sort n items in less than $n \log n$ running time. There are also some special cases when it's possible to sort with running time n. We won't cover these algorithms here, though.

Algorithms that sort in $n \log n$ steps are fairly complicated and will have to wait for another course. So will a proof of why that's the best you can do. If you're really interested in $n \log n$ sorting, look for *mergesort* or *quicksort* algorithms.

You probably have some idea of how to sort. Suppose you're given a pile of a dozen exam papers and are asked to put them in order by the students' names. Many people would do something like this:

1. Flip through the pile and find the paper that should go first.
2. Put that paper face-down on the table.
3. Repeat from step 1 with the remaining pile, until there are no papers left.

This method is roughly equivalent to the *selection sort* algorithm that can be used on a list. The idea behind selection sort is to scan through a list for the smallest element and swap it with the first item.

So, if we look at the list 6, 2, 8, 4, 5, 3, a selection sort will do the following operations to get it in order. At each step, the parts of the list that we *know* are sorted are bold.

Iteration	**Initial List**	**Operation**
1.	6, 2, 8, 4, 5, 3	swap 2 and 6
2.	**2**, 4, 8, 6, 5, 3	swap 3 and 4
3.	**2**, **3**, 8, 6, 5, 4	swap 4 and 8
4.	**2**, **3**, **4**, 6, 5, 8	swap 6 and 5
5.	**2**, **3**, **4**, **5**, 6, 8	swap 6 and 6 (do nothing)
6.	**2**, **3**, **4**, **5**, **6**, 8	swap 8 and 8 (do nothing)
	2, **3**, **4**, **5**, **6**, **8**	

Pseudocode of the selection sort algorithm can be found in Figure 6.5. A Python implementation can be found in Figure 6.6. This algorithm has running time n^2.

If you count the number of times the inner loop runs for each element, you find that it takes n steps for the first element, then $n-1$ for the second,

for every element e from the list,
    for every element f from e to the end of the list,
        if $f < smallest$,
            set *smallest* to f
    swap *smallest* and e

Figure 6.5: Algorithm for selection sort

then $n-2, \ldots, 2, 1$. So, the total number of steps is

$$\sum_{i=1}^{n} i = \frac{n(n-1)}{2} = n^2/2 - n/2\,.$$

Removing lower-order terms and constants, you can see that the running time is n^2.

Selection sort will be quite slow for large lists; one of the $n \log n$ algorithms should be used instead.

Sorting is covered in more detail in CMPT 125, 225, and 307. As you progress, the focus is less on sorting itself and more on how it is used to solve other problems.

Topic 6.3 Recursion

As we have seen many times, it's possible for a function to call another function. For example, in Figure 6.1, the `search` function uses both `range` and `len` to create the appropriate loop.

But, it's also possible for a function to call *itself*. This is useful when a problem can be solved by breaking it into parts and solving the parts. This technique is called *recursion*, and is very important since many algorithms are most easily described recursively.

For example, consider calculating the *factorial* of a number. The factorial of n is usually written $n!$ and is the product of the numbers from 1 to n: $1 \times 2 \times 3 \times \cdots \times (n-1) \times n$. The factorial function is often defined in terms of itself:

$$n! = \begin{cases} 1 & \text{for } n = 0 \\ n \times (n-1)! & \text{for } n > 0 \end{cases}$$

```
def selection_sort(lst):
    """
    Sort lst in-place using selection sort
    """
    for pos in range(len(lst)):
        # get the next smallest in lst[pos]

        # find the next smallest
        small = lst[pos] # smallest value seen so far
        smallpos = pos   # position of small in lst
        for i in range(pos+1, len(lst)):
            # check each value, searching for one
            # that's smaller than the current smallest.
            if lst[i] < small:
                small = lst[i]
                smallpos = i

        # swap it into lst[pos]
        lst[pos], lst[smallpos] = lst[smallpos], lst[pos]
```

Figure 6.6: Python implementation of selection sort

```
def factorial(n):
    """
    Calculate n! recursively.

    >>> factorial(10)
    3628800
    >>> factorial(0)
    1
    """

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

Figure 6.7: Recursively calculate factorials

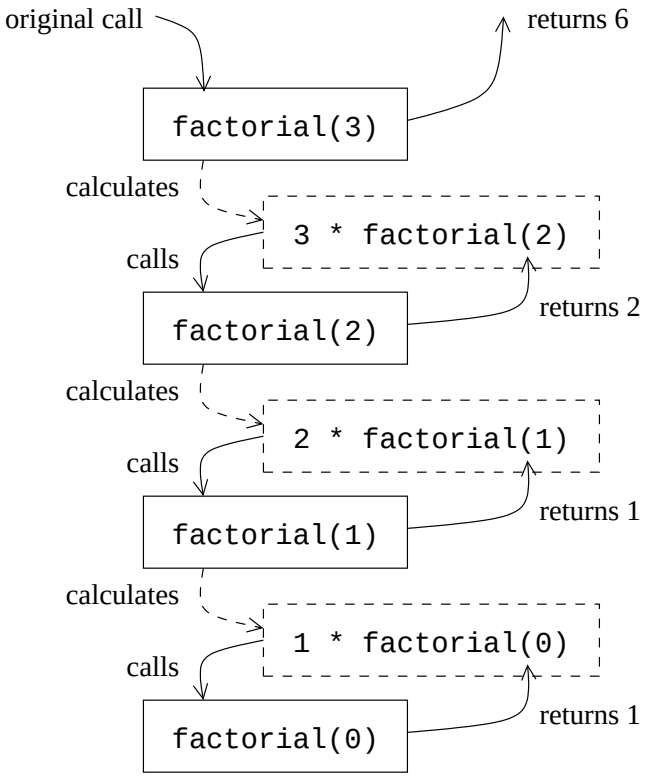

Figure 6.8: Functions calls made to calculate `factorial(3)`

We can use this same definition to create a Python function that calculates the factorial of a (positive) integer.

The code in Figure 6.7 defines a function that correctly calculates $n!$. It does this by calling *itself* to calculate the factorial of $n - 1$.

How It Works

Whenever a function calls itself, you should think of a new copy of the function being made. For example, if we call `factorial(3)`, while running, it will call `factorial(2)`. This will be a separate function call, with separate arguments and local variables. It will run totally independently of the other instance of the function.

Figure 6.8 contains a diagram representing all of the calls to the `factorial` function required to calculate `factorial(3)`. As you can see, `factorial(3)`

calls `factorial(2)`, which itself calls `factorial(1)`, which calls `factorial(0)`.

Because of the way the `factorial` function is written, `factorial(0)` returns one. Then, `factorial(1)` can complete its calculations and return; then `factorial(2)` can finish. Finally, `factorial(3)` completes and returns the result originally requested.

Once everything is put together, the function actually computes the correct value. This is made possible by the design of the function: as long as the recursive calls return the right value, it will calculate the correct result for `n`.

The idea that the function calls itself, or that many copies of the function are running at one time will probably seem strange at first. What you really need to remember is simple: it works. Python can keep track of several instances of the same function that are all operating simultaneously, and what each one is doing.

Understanding Recursion

When looking at a recursive algorithm, many people find it too complicated to think of every function call, like in Figure 6.7. Keeping track of every step of the recursion isn't really necessary to believe that the function works, or even to design one yourself.

When reading a recursive function, you should just assume that the recursive calls return the correct value. In the example, if you take on faith that `factorial(n-1)` returns $(n-1)!$ correctly, then it's clear that the function does the right thing in each case and actually calculates $n!$.

You can let the programming language take care of the details needed to run it.

If you look at Figure 6.7 and assume that the recursive call works, then it's clear that the whole function does as well. The key to the logic here is that the recursive calls are to *smaller* instances of the same problem. As long as the recursive calls keep getting smaller, they will eventually hit the $n = 0$ case and the recursion will stop.

When writing recursive functions, you have to make sure that the function and its recursive calls are structured so that analogous logic holds. Your function must make recursive calls that head towards the *base case* that ends the recursion.

We will discuss creating recursive functions further in the next topic.

Students who have taken MACM 101 may recognize this as being very similar to proofs by induction. The ideas are very similar: we need somewhere to start (base case), and some way to take a step to a smaller case (recursive/inductive case). Once we have those, everything works out.

Topic 6.4 Designing with Recursion

As an example of a recursive function, let's look at the problem of reversing a string. We will construct a recursive function that does this, so calling `reverse("looter")` should return `"retool"`.

In order to write a recursive function to do this, we need to view the problem in the right way and create a recursive function that implements our recursive algorithm.

Step 1: Find a Smaller Subproblem

The whole point of making a recursive call is to solve a similar, but smaller problem. If we have decomposed the problem properly, we can use the recursive solution to build a solution to the problem we are trying to solve.

In the factorial example, this relies on noticing that $n! = n \times (n-1)!$ (in almost all cases). Then, if we can somehow calculate $(n-1)!$ it's easy to use that to calculate $n!$.

To reverse the string, we have to ask: if we reverse part of the string, can we use that to finish reversing the whole string? If we reverse the tail of the string (`string[1:]`, everything except the first character), we can use it.

For example, if we are trying to reverse the string `"looter"`, the tail (`string[1:]`) is or `"ooter"`. If we make a recursive call to reverse this (`reverse(string[1:])`), it should return `"retoo"`. This is a big part of the final solution, so the recursive call will have done useful work. We can later use this to build the reverse of the whole string.

Step 2: Use the Solution to the Subproblem

Once you have taken the problem you're given and found a subproblem that you can work with, you can get the result with a recursive call to the function.

In the factorial example, we know that once we have calculated $(n-1)!$, we can simply multiply by n to get $n!$. In the Python implementation, this becomes `n * factorial(n-1)`. If we put our faith in the correctness of the calculation `factorial(n-1)`, then this is definitely the correct value to return.

When reversing a string, if we reverse all but the first character of the string (`reverse(string[1:])`), we are very close to the final solution. All that remains is to put the first character (`string[0]`) on the end.

Again using `"looter"` as an example, `reverse(string[1:])` returns `"retoo"` and `string[0]` is `"l"`. The whole string reversed is:

```
reverse(string[1:]) + string[0]
```

This evaluates to `"retool"`, the correct answer. In general, this expression gives the correct reversal of `string`.

Again, in both examples the method is the same: make a recursive call on the subproblem, and use its result to construct the solution you have been asked to calculate.

Step 3: Find a Base Case

There will be a few cases where the above method won't work. Typically these will be the smallest cases where it's not possible to subdivide the problem further.

These cases can simply be handled with an `if` statement. If the arguments point to a base case, return the appropriate result. Otherwise, proceed with the recursive solution as designed above.

In the factorial example, the identity $n! = n \times (n-1)!$ isn't true for $n = 0$. In this case, we know the correct answer from the definition of factorial: $0! = 1$. In the implementation, we check for this case with an `if` statement and return the correct result. In all other cases, the recursive solution is correct, so we use it.

For reversing a string, there is also one case where the method outlined above for the recursive case can't be followed. Again, we look at the smallest cases of the problem, which cannot be subdivided further.

What should be the result when we call `reverse("")`? Presumably, the reverse of the empty string is `""`, but the above method won't give this result. In fact, if we try to extract element 0 from this string, it will cause an `IndexError` since it doesn't have a zero-th character. Since this case

doesn't match the recursive algorithm, it will be our base case and handled separately.

We should also check other small cases: what is the reverse of a single-character string? The function call `reverse("X")` *should* return `"X"`. We can check our recursive method for this case, when `string` is `"X"`:

```
reverse(string[1:]) + string[0] == reverse("") + "X"
```

Since we just decided that `reverse("")` will return `""`, this becomes `"" + "X"`, which is `"X"`. This is the correct result, so we don't have to worry about single-character strings as a base case: the recursive case already handles them correctly.

It's very important every recursive call will *eventually* get to a base case. The recursive call that is made in the function must be at least one step closer to the base case: the part we decide to solve recursively is *smaller* than the original problem.

Remember that the base case is where the recursion will end. Once it does, the base case will return the correct result (since we made sure it did). From there, the recursive calls will begin to "unwind", with each one returning the correct result for the arguments it was given.

If this isn't the case, the function will keep making more and more recursive calls without ever stopping. This is called *infinite recursion*. Look back at Figure 6.8 and imagine what it would look like if we didn't have the special case for `num==0`. It would keep making more and more calls until the program was halted. This is analogous to the infinite loops you can create with `while`.

Python will stop when the recursion passes a certain "depth". It will give the error "maximum recursion depth exceeded". If you see this, you have probably created infinite recursion.

Step 4: Combine the Base and Recursive Cases

Once we have identified the base case(s) and what recursive calculation to do in other cases, we can write the recursive function.

Assembling the parts is relatively easy. In the function, first check to see if the argument(s) point to a base case. If so, just return the solution for this case. These should be very easy since the base cases are the smallest possible instances of the problem.

if we have a base case,

    return the base case solution

otherwise,

    set *rec_result* to the result of a recursive call on the subproblem
    return the solution, built from *rec_result*

Figure 6.9: Pseudocode for a recursive algorithm

If we don't have a base case, then the recursive case applies. We find our subproblem and call the same function recursively to solve it. Once that's done, it should be possible to transform that into the solution we need. Once we know it, it will be returned.

Pseudocode for a recursive function can be found in Figure 6.9. Compare this with the implementation of `factorial` in Figure 6.7. Figure 6.7 doesn't put the recursive result in a variable because the calculation is so short, but it's otherwise the same.

The example of reversing a string has been implemented in Figure 6.10. It uses the parts constructed above and the outline in Figure 6.9. It does correctly return the reverse of any string.

Debugging Recursion

Debugging a recursive function can be trickier than non-recursive code. In particular, when designing the recursive function, we made an assumption: that the recursive call to the same function returned the correct result. Since the function is relying on itself working, tracing problems can be difficult.

The key to finding errors in recursive code is to first side-step this problem. There are cases in the recursive function that don't make recursive calls: the base cases. This gives us somewhere to start testing and debugging.

The first thing that should be done when testing or trying to find errors is to examine the base case(s). For example, in the above examples, the first things to test would be:

```
>>> factorial(0)
1
>>> reverse("")
''
```

```
def reverse(string):
    """
    Return the reverse of a string

    >>> reverse("bad gib")
    'big dab'
    >>> reverse("")
    ''
    """
    if len(string)==0:
        # base case
        return ""
    else:
        # recursive case
        rev_tail = reverse(string[1:])
        return rev_tail + string[0]
```

Figure 6.10: Recursively reversing a string

If the base cases work correctly, then at least we have that to work with. If not, we know what code to fix.

Once we know the base cases are working, we can then easily test the *cases that call the base case*. That is, we now want to look at the arguments to the function that are one step away from the base case. For example,

```
>>> factorial(1)
1
>>> reverse("X")
'X'
>>> reverse("(")
'('
```

In each case here, the recursive code runs once, and it calls the base case. We can manually check the calculations in these cases and confirm that our expectations match the results.

If these aren't correct, then there is something wrong with the way the recursive code uses the base case (which we have already tested) to compute the final solution. There are a couple of possibilities here: the wrong recursive

call might be made, or the calculation done with the recursive result could be incorrect. Both of these should be checked and corrected if necessary.

If these cases work, but the overall function still isn't correct, you can proceed to cases that are another step removed (two steps from the base case). For example, `factorial(2)` and `reverse("Ab")`. You should quickly find some case that is incorrect and be able to diagnose the problem.

Another Example

As a final example of recursion, we will create a function that inserts spaces *between* characters in a string and returns the result. The function should produce results like this:

```
>>> spacedout("Hello!")
'H e l l o !'
>>> spacedout("g00DbyE")
'g 0 0 D b y E'
```

We can work through the steps outlined above to create a recursive solution.

1. [Find a Smaller Subproblem.] Again, we look for cases that are one "step" smaller and can contribute to an overall solution. Here, we will take all but the *last* character of the string, so for `"marsh"`, we will take the substring `"mars"`.

   The recursive call will be `spacedout(string[:-1])`.

2. [Use the Solution to the Subproblem.] If the recursive is working correctly, we should get a spaced-out version of the substring. In the example, this will be `"m a r s"`. We can use this to finish by adding a space and the last character to the end.

   So, the final result will be `rec_result + " " + string[-1]`.

3. [Find a Base Case.] The above method clearly won't work for the empty string since there is no last character to remove. In this case, we can just return the empty string.

   But, the above also won't work for strings with a single character. The recursive method applied to `"X"` will return `" X"`, with an extra space at the beginning. We will have to handle these as a base case as well: the correct result is to return the same string unchanged.

```
def spacedout(string):
    """
    Return a copy of the string with a space between
    each character.

    >>> spacedout("ab cd")
    'a b   c d'
    >>> spacedout("")
    ''
    >>> spacedout("Q")
    'Q'
    """
    if len(string) <= 1:
        return string
    else:
        head_space = spacedout(string[:-1])
        return head_space + " " + string[-1]
```

Figure 6.11: Recursively putting spaces between characters in a string

In fact, these two cases can easily be combined. If the string has 0 or 1 characters, it can be returned unchanged.

4. [Combine the Base and Recursive Cases.] The above work has been combined in the implementation in Figure 6.11.

Check-Up Questions

- Repeat the spaced-out string example using everything but the *first* character as the subproblem. That is, for the string `"hearth"`, the subproblem should be `"earth"`, instead of `"heart"`. You should be able to get a different recursive function that produces the same results.
- Write a recursive function `list_sum(lst)` that adds up the numbers in the list given as its argument.
- Write a recursive function `power(x,y)` that calculates x^y, where y is a positive integer. To create a subproblem, you will have to decrease *one of* `x` or `y`. Which one gets you a useful result?

Topic 6.5 What isn't computable?

Throughout this course, we have solved many problems by creating algorithms and implementing the algorithms in Python. But, there is a fundamental question here: are there problems that you can't write a computer program to solve?

The answer is yes. It's possible to prove that, for some problems, it's impossible to write a program to solve the problem. That means that the lack of a solution isn't a failing of the programming language you're using, your programming abilities, or the amount of time or money spent on a solution. There is no solution to these problems because it's fundamentally impossible to create one with the tools we have available to do computations.

The Halting Problem

For example consider the following problem:

> Consider a particular program. Given the input given to the program, determine whether or not the program will ever finish.

This problem is called the *halting problem*. The basic idea is to decide whether or not a program "halts" on particular input. Does it eventually finish or does it go into an infinite loop?

This is something that would be very useful to be able to compute. The Python environment could have this built in and warn you if your program was never going to finish. It would be very helpful when it comes to debugging.

Unfortunately, it's *impossible* to write a program that looks at any program to determine whether or not it halts.

Suppose someone comes along who claims to have created a function `halts(prog, input)` that solves the halting problem: it returns `True` if the program in the file `prog` halts on the given input and `False` if not. You write the program in Figure 6.12 and ask if you can test their function.

This program is based on asking the `halts` function what happens when *a program is given itself as input*. So, we imagine what would happen if you ran a program and then typed its filename and pressed enter. Maybe that isn't sensible input for the program but that doesn't matter: we only care if the program halts or not, not if it does something useful.

```
prog = raw_input("Program file name: ")

if halts(prog, prog):
    while True:
        print "looping"
else:
    print "done"
```

Figure 6.12: Fooling a function that claims to solve the halting problem.

Does the program in Figure 6.12 halt? Sometimes. If its input is a program that halts when given itself as input, Figure 6.12 enters an infinite loop. If the input program doesn't halt, then the program in Figure 6.12 stops immediately.

Suppose we run the program in Figure 6.12 and give it itself as input. So, we enter the file name of the program in Figure 6.12 at the prompt. What answer does the program give?

Without knowing anything about the `halts` function, we know that it will always give the wrong answer for these inputs. These are the two possibilities:

1. `halts("fig6.12.py", "fig6.12.py")` returns `True`. Then the program in Figure 6.12 would have entered an infinite loop: it should have returned `False`.
2. `halts("fig6.12.py", "fig6.12.py")` returns `False`. Then the program in Figure 6.12 would have printed one line and stopped: it should have returned `True`.

So, no matter what claims are made, a program that claims to compute the halting problem will *always* make some mistakes.

The idea of feeding a program itself as input might seem strange at first, but it's a perfectly legitimate thing to do. It should be enough to convince you that there are *some* inputs where a halting function fails. There will be many more for any particular attempt.

There's nothing that says you can't write a program that answers the halting problem correctly *sometimes*. The point is that the problem in general is impossible to solve with any computer.

Virus Checking

One problem that you may have had to deal with in the real-world is keeping viruses away from your computer.

A computer *virus* is a program that is designed to spread itself from one location to another—between different programs and computers. There are many programs that scan your computer for viruses and report any problems.

All of these programs require an up to date list of virus definitions. Every week or so, they download a new list of viruses over the Internet. These "definitions" contain information like "files that contain data like this... are infected with virus X."

You may have wondered why this is necessary: why can't virus checkers just look for programs that behave like viruses. By definition, a virus has to be a program that reproduces; just look for programs that put a copy of themselves into another program.

But again, we run into a problem of computability. Writing a program to check for programs that reproduce themselves isn't possible. So, it's impossible to write a perfect anti-virus program. The most effective solution seems to be the creation of lists of viruses and how to detect them. A fairly simple program can scan a hard drive looking for *known* virus signatures.

The downside is that the program will only detect known viruses. When new viruses are created, they must be added to the list. As a result, it's necessary to update virus definition files regularly.

Again, remember that these problems (halting and virus checking) aren't impossible to compute because of some limitation in a programming language or the computer you're working with. They are impossible with *every* computational device anybody has ever thought of.

Summary

This unit covers several important aspects of algorithms and computability. Sorting, searching, and recursion are important techniques for designing efficient algorithms; you will see much more of them if you go on in Computing Science.

The uncomputability topic is important to the way problems are solved with computer. The same problems can be solved with any computer and there are some problems that can't be solved with any computer.

Key Terms

- searching
- linear search
- binary search
- sorting
- selection sort
- recursion
- base case
- uncomputable
- halting problem
- virus checking