*7.5 DESIGNING EFFICIENT ALGORITHMS

For most of the problems we have encountered so far, we have really only considered one algorithm. But for most real problems that involve real data, there are many possible algorithms with varying time complexities, and a little more ingenuity is required to find the best one.

Removing duplicates

To illustrate, let's consider a slightly more involved problem that can be formulated in terms of the following "real world" situation. Suppose you organized a petition drive in your community, and have now collected all of the lists of signatures from your volunteers. Before you can submit your petition to the governor, you need to remove duplicate signatures from the combined list. Rather than try to do this by hand, you decide to scan all of the names into a file, and design an algorithm to remove the duplicates for you. Because the signatures are numbered, you want the final list of unique names to be in their original order.

As we design an algorithm for this problem, we will keep in mind the four steps outlined in Chapter 1:

1. Understand the problem.
2. Design an algorithm.
3. Write a program.
4. Look back.

Reflection 1 *Before you read any further, make sure you understand this problem. If you were given this list of names on paper, what algorithm would you use to remove the duplicates?*

The input to our problem is a list of items, and the output is a new list of unique items in the same order they appeared in the original list. We will start with an intuitive algorithm and work through a process of refinement to get progressively better solutions. In this process, we will see how a critical look at the algorithms we write can lead to significant improvements.

A first algorithm

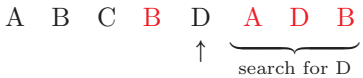
There are several different approaches we could use to solve this problem. One intuitive idea is to iterate over the items and mark any duplicates of each one found further down the list. Once all of the duplicates are marked, we can remove them from a copy of the original list. The following example illustrates this approach with a list containing four unique names, abbreviated A, B, C, and D. The algorithm starts at the beginning of the list, which contains the name A. Then we search down the list and mark the duplicate in red.

Next we move on to the second item, B, and mark its duplicates.

Some items, like the next item, C, do not have any duplicates.

The item after C is already marked as a duplicate, so we skip the search.

The next item, D, is not marked so we search for its duplicates down the list.



Finally, we finish iterating over the list, but find that the remaining items are already marked as duplicates.

Once we know where all the duplicates are, we can make a copy of the original list and remove the duplicates from the copy. This algorithm is written in pseudocode below.

Algorithm REMOVE DUPLICATES — DRAFT 1

Input: a list of *data*

```
1  repeat for each item in data:
2      if item is not marked as a duplicate, then:
3          mark later copies of item as duplicates
4  unique items ← a copy of data
5  repeat for each item in data:
6      if item is marked, remove it from unique items
```

Output: *unique items*

Let's think about how to implement this algorithm in Python.

Reflection 2 *How are we going to "mark" items in a Python list?*

We can't really directly "mark" items in a Python list, but we can remember the indices of "marked" items in a separate *duplicates* list. So line 2 of the algorithm will need to check whether the index of the current item is in this *duplicates* list, and line 3 will need to add indices to the *duplicates* list.

Reflection 3 *Given this manner of "marking" items in Python, will the* `for` *loop on line 2 need to iterate over list items or the indices of the list items?*

Since the algorithm will be working with indices of items, the loop will need to iterate over the indices of the items in *data*. Let's update our pseudocode algorithm to reflect this so that it will be easier to implement in Python.

Algorithm REMOVE DUPLICATES — DRAFT 2

Input: a list of *data*

1 *duplicates* ← an empty list of indices
2 repeat for each *index* of an item in *data*:
3 if the *index* is not in *duplicates*, then:
4 add the indices of later copies of *data*[*index*] to *duplicates*
5 *unique items* ← a copy of *data*
6 repeat for each *index* in *duplicates*:
7 remove the item in position *index* from *unique items*

Output: *unique items*

Now it should be pretty clear how we are going to implement lines 1–2. To implement line 3, we need to search for an item in a list. We could use the Python `in` operator, but let's instead revisit how to write a search function from scratch. Recall that, in Section 6.4, we developed a linear search algorithm (named `find`) that returns the index of the first occurrence of a substring in a string. You may have applied this same idea to implement the following `linearSearch` function in Exercise 7.1.17.

```
def linearSearch(data, target):
    """Find the index of the first occurrence of target in data.

    Parameters:
        data:   a list to search in
        target: a value to search for

    Return value: the index of the first occurrence of target in data
    """

    for index in range(len(data)):
        if data[index] == target:
            return index
    return -1
```

The function iterates over the indices of the list named `data`, checking whether each item equals the target value. If a match is found, the function returns the index of the match. Otherwise, if no match is ever found, the function returns −1. Assuming we use the same variable names in our Python function, we can use this function to implement line 3 like this:

```
if linearSearch(duplicates, index) == -1:   # index not in duplicates
```

To implement line 4, we need to find *all* of the indices of items equal to `data[index]` that occur *later* in the list. A function to do this will be very similar to `linearSearch`, but will differ in two ways. First, it must return a list of indices instead of a single index. Second, it will require a third parameter that specifies where the search should begin.

```
def linearSearchAll(data, target, start):
    """Find the indices of all occurrences of target in data.

    Parameters:
        data:   a list to search in
        target: a value to search for
        start:  the index in data to start searching from

    Return value: a list of indices of all occurrences of target in data
    """

    foundIndices = [ ]
    for index in range(start, len(data)):
        if data[index] == target:
            foundIndices.append(index)
    return foundIndices
```

With this new function, we can implement line 4 like this:

```
positions = linearSearchAll(data, data[index], index + 1)
duplicates.extend(positions)
```

The first statement gets a list of indices of all items matching `data[index]`, starting at position `index + 1`. The second statement adds these positions to the `duplicates` list. The `extend` method takes a list as an argument and appends each item in this list to the list on which it is operating.

Reflection 4 *How does the statement* `duplicates.extend(positions)` *differ from* `duplicates.append(positions)`*?*

Once we have the list of duplicates, we remove them from the *unique items* list in lines 6–7. The way the algorithm is written suggests using the `pop` method, since `pop` takes an index as a parameter. But, as we saw in Section 7.2, using `pop` in a loop can be tricky. An alternative approach would be to use a list accumulator to build the new list up from an empty list. To do this, we will need to iterate over the original list and append items to the new list only if their indices are not in the list of duplicate indices, like this:

```
uniqueItems = [ ]
for index in range(len(data)):
    if linearSearch(duplicates, index) == -1:  # index not in duplicates
        uniqueItems.append(data[index])
```

Putting these pieces together, we have the following Python function.

```
def removeDuplicates1(data):
    """Return a list of the unique items in data, in their original order.

    Parameter:
        data: a list

    Return value: a new list of unique items
    """
```

```
    duplicates = [ ]
    for index in range(len(data)):
        if linearSearch(duplicates, index) == -1: # index not in duplicates
            positions = linearSearchAll(data, data[index], index + 1)
            duplicates.extend(positions)

    uniqueItems = [ ]
    for index in range(len(data)):
        if linearSearch(duplicates, index) == -1: # index not in duplicates
            uniqueItems.append(data[index])

    return uniqueItems
```

Reflection 5 *Write some unit tests for the new function on a variety of inputs.*

This algorithm works, but that does not mean that we should immediately leave this problem behind. In addition to testing, it is important that we take some time to critically "look back" and reflect on the quality of our solution. Can the function be simplified or made more efficient? What is its time complexity?

Reflection 6 *The function above can be simplified a bit. Look at the similarity between the two* `for` *loops. Can they be combined?*

The two `for` loops can, in fact, be combined. If the condition in the first `if` statement is true, then this must be the first time we have seen this particular list item. Therefore, we can append it to `uniqueItems` right then. The resulting function is a bit more streamlined:

```
1 def removeDuplicates2(data):
2     """ (docstring omitted) """

3     duplicates = [ ]
4     uniqueItems = [ ]
5     for index in range(len(data)):
6         if linearSearch(duplicates, index) == -1: # index not in duplicates
7             positions = linearSearchAll(data, data[index], index + 1)
8             duplicates.extend(positions)
9             uniqueItems.append(data[index])

10    return uniqueItems
```

Now let's analyze the worst case asymptotic time complexity of the algorithm. As usual, we will call the length of the input list n (i.e., `len(data)` is n). The statements on lines 3, 4, and 10 each count as one elementary step. The rest of the algorithm is contained in the `for` loop, which iterates n times. So the `if` statement on line 6 is executed n times and, in the worst case, the statements on lines 7–9 are each executed n times as well. But how many elementary steps are hidden in each of these statements?

Reflection 7 *How many elementary steps are required in the worst case by the call to* `linearSearch` *on line 6? (You might want to refer back to page O6.7-3, where we talked about the time complexity of a linear search.)*

We saw in Section 6.7 that linear search is a linear-time algorithm (hence the name). So

Tangent 1: Amortization applied to append

Consider a typical call to the `append` method, like `data.append(item)`. How many elementary steps are involved in this call? We have seen that some Python operations, like assignment statements, require a constant number of elementary steps while others, like the `count` method, can require a linear number of elementary steps. The `append` method is an interesting case because the number of elementary steps depends on how much space is available in the chunk of memory that has been allocated to hold the list. If there is room available at the end of this chunk of memory, then a reference to the appended item can be placed at the end of the list in constant time. However, if the list is already occupying all of the memory that has been allocated to it, the `append` has to allocate a larger chunk of memory, and then copy the entire list to this larger chunk. The amount of time to do this is proportional to the current length of the list. To reconcile these different cases, we can use the average number of elementary steps over a long sequence of `append` calls, a technique known as *amortization*.

When a new chunk of memory needs to be allocated, Python allocates more than it actually needs so that the next few `append` calls will be quicker. The amount of extra memory allocated is cleverly proportional to the length of the list, so the extra memory grows a little more each time. Therefore, a sequence of `append` calls on the same list will consist of sequences of several quick operations interspersed with increasingly rare slow operations. If you take the average number of elementary steps over all of these appends, it turns out to be a constant number (or very close to it). Thus an `append` can be considered one elementary step!

the number of elementary steps required by line 6 is proportional to the length of the `duplicates` list. The length of this list will be zero in the first iteration of the loop, but may contain as many as $n-1$ indices in the second iteration. So the total number of elementary steps in each of these later iterations is $n-1$ in the worst case.

Reflection 8 *Why could* `duplicates` *have length* $n-1$ *after the first iteration of the loop? What value of* `data` *would make this happen?*

Therefore, in the worst case, the total number of elementary steps executed in line 6, over all the iterations of the loop, is at most $1+(n-1)(n-1)$: one in the first iteration of the loop, then at most $n-1$ for each iteration thereafter. This simplifies to n^2-2n+2 which, asymptotically, is $\mathcal{O}(n^2)$.

The `linearSearchAll` function called on line 7 is a lot like a regular linear search, but it only iterates from `start` to the end of the list and it contains an `append` in the body of the loop. Therefore, since we can consider the `append` to be an elementary step (see Tangent 1), `linearSearchAll` requires about $n-$`start` elementary steps instead of n. When `linearSearchAll` is called on line 7, the value of `index + 1` is passed in for `start`, so it really takes $n-$(`index + 1`) steps. When `index` is 0, $n-$(`index + 1`) $= n-1$. When `index` is 1, $n-$(`index + 1`) $= n-2$. And so on. So the total number of elementary steps executed in line 7 is $(n-1)+(n-2)+\cdots+1$. We have seen this sum before (see Tangent 4.1); it is equal to the triangular number

$$\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

which is also $\mathcal{O}(n^2)$.

Finally, lines 8 and 9 involve a total of $\mathcal{O}(n)$ elementary steps. This is easier to see with line

9 because it involves at most one `append` per iteration. The `extend` method called on line 8 effectively appends all of the values in `positions` to the end of `duplicates`. Although one call to `extend` may append more than one index to `duplicates` at a time, overall, each index of `data` can be appended at most once, so the total number of elementary steps over all of the calls to `extend` must be proportional to n. In summary, we have determined the following numbers of elementary steps for each line.

Line	Elementary steps
3	$\mathcal{O}(1)$
4	$\mathcal{O}(1)$
5	$\mathcal{O}(n)$
6	$\mathcal{O}(n^2)$
7	$\mathcal{O}(n^2)$
8	$\mathcal{O}(n)$
9	$\mathcal{O}(n)$
10	$\mathcal{O}(1)$

Since the maximum number of elementary steps for any line is $\mathcal{O}(n^2)$, we have a *quadratic-time algorithm*.

A more elegant algorithm

In our current algorithm, we collect a list of indices of duplicate items, and search this list before deciding whether to append a new item to the growing list of unique items. But, since we are now constructing the list of unique items in the same `for` loop, we could decide whether the current item (`data[index]`) is a duplicate by searching for it in `uniqueItems` instead. This change eliminates the need for the `duplicates` list altogether and greatly simplifies the algorithm:

```
def removeDuplicates3(data):
    """ (docstring omitted) """

    duplicates = [ ]
    uniqueItems = [ ]
    for index in range(len(data)):
        if linearSearch(uniqueItems, data[index]) == -1:
            positions = linearSearchAll(data, data[index], index + 1)
            duplicates.extend(positions)
            uniqueItems.append(data[index])

    return uniqueItems
```

In addition, since we are not storing indices any longer, we can iterate over the items in the list instead of the indices, giving a much cleaner look to the function:

```
1 def removeDuplicates4(data):
2     """ (docstring omitted) """

3     uniqueItems = [ ]
4     for item in data:
5         if linearSearch(uniqueItems, item) == -1:
6             uniqueItems.append(item)

7     return uniqueItems
```

Reflection 9 *This revised algorithm is certainly more elegant. But is it more efficient?*

To answer this question, let's revisit our time complexity analysis. The `for` loop still iterates n times and, in the worst case, both the call to `linearSearch` and `append` are still executed in every iteration of the loop. We saw above that the number of elementary steps executed by the `linearSearch` function depends on the length of the list that is passed in. In this case, `uniqueItems` can be no longer than the number of previous iterations, since at most one item is appended to it in each iteration. So the length of `uniqueItems` can grow by at most one in each iteration, meaning that the number of elementary steps executed by `linearSearch` can also grow by one in each iteration in the worst case. So the total number of elementary steps executed by all of the calls to `linearSearch` in the worst case is

$$1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

or, once again, $\mathcal{O}(n^2)$. In summary, the numbers of elementary steps for each line are now:

Line	Elementary steps
3	$\mathcal{O}(1)$
4	$\mathcal{O}(n)$
5	$\mathcal{O}(n^2)$
6	$\mathcal{O}(n)$
7	$\mathcal{O}(1)$

Like our previous algorithm, the maximum value in the table is $\mathcal{O}(n^2)$, so our new algorithm is still a quadratic-time algorithm.

A more efficient algorithm

Reflection 10 *Do you think it is possible to design a more efficient algorithm for this problem? If so, what part(s) of the algorithm would need to be made more efficient? Are there parts of the algorithm that are absolutely necessary and therefore cannot be made more efficient?*

To find all of the duplicates in a list, it seems obvious that we need to look at every one of the n items in the list (using a `for` loop or some other kind of loop). Therefore, the time complexity of any algorithm for this problem must be at least $\mathcal{O}(n)$, or linear-time. But is a linear-time algorithm actually possible? Apart from the loop, the only work-intensive component of the algorithm remaining is the linear search in line 5. Can the efficiency of this step be improved?

Searching for data efficiently, as we do in a linear search, is of utmost importance in computer

science. There are a wide variety of innovative ways to store data to facilitate fast access, most of which are beyond the scope of an introductory book. However, we have already seen one alternative. Recall from Tangent 7.2 that a dictionary is cleverly implemented so that access can be considered a constant-time operation. So if we can store information about duplicates in a dictionary, we can perform the search in line 5 in constant time!

The trick is to store the items that we have already seen as keys in a dictionary instead of items in a list. Then we can test whether we have already seen the item by checking whether `item` is already a key in the dictionary. The new function with these changes follows.

```
1 def removeDuplicates5(data):
2     """ (docstring omitted) """

3     uniqueItems = { }
4     for item in data:
5         if item not in uniqueItems:
6             uniqueItems[item] = True

7     return list(uniqueItems.keys())
```

In our new function, we associate the value `True` with each key, but this is an arbitrary choice because we never actually use these values. At the end of the function, we return the list of keys, which are exactly the unique items in the list.

Since every statement in the body of the `for` loop is now one elementary step, lines 5 and 6 are now executed a total of of $\mathcal{O}(n)$ times, as summarized below.

Line	Elementary steps
3	$\mathcal{O}(1)$
4	$\mathcal{O}(n)$
5	$\mathcal{O}(n)$
6	$\mathcal{O}(n)$
7	$\mathcal{O}(1)$

Therefore, the `removeDuplicates5` function is a linear-time algorithm. As we saw in Section 6.7, this makes a significant difference! Exercise 7.5.4 asks you to investigate this difference experimentally for yourself.

Exercises

7.5.1* The `if` statement in the `removeDuplicates5` function can actually be completely removed and the algorithm will still be correct. Explain why.

7.5.2. When analyzing a text, one measure of its complexity is the number of unique words it contains, called its *vocabulary*. Write a function

```
vocabulary(text)
```

that returns the vocabulary of the string `text`. Use the `wordTokens` function from Section 6.1.

7.5.3* Write an algorithm that returns a list of only those items in `data` that are duplicates. For example, if `data` were `[1, 2, 3, 1, 3, 1]`, the function should return the list `[1, 3]`.

7.5.4. In this exercise, you will write a program that tests whether the linear-time `removeDuplicates5` function really is faster than the quadratic-time `removeDuplicates2` and `removeDuplicates4` functions. First, write a function that creates a list of n random integers between 0 and $n - 1$ using a list accumulator and the `random.randrange` function. Then write another function that calls each of the three functions with this list as the argument. Time how long each call takes using the `time.time` function, which returns the current time in elapsed seconds since a fixed "epoch" time (usually midnight on January 1, 1970). By calling `time.time` before and after each call, you can find the number of seconds that elapsed. Repeat your experiment with $n = 100, 1000, 10{,}000$, and 100,000 (this will take a long time). Describe your results.

7.5.5. In a round-robin tournament, every player faces every other player exactly once, and the player with the most head-to-head wins is deemed the champion. The following algorithm simulates a round-robin tournament.

Algorithm ROUND ROBIN

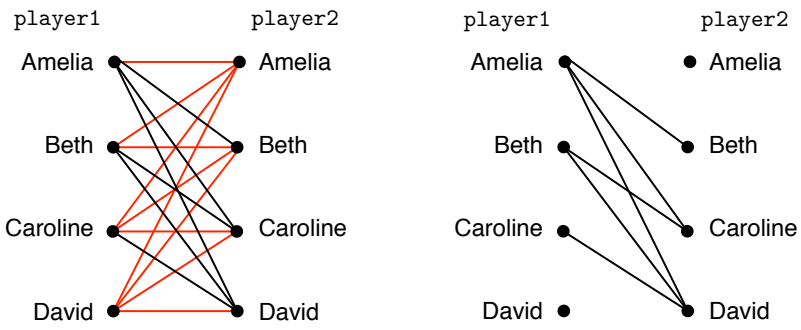
Input: a list of *n players*

```
1  repeat for each player in players:
2      wins[player] ← 0
3  repeat for each player1 in players:
4      repeat for each player2 in players:
5          if player2 ≠ player1:
6              winner ← winner of matchup between player1 and player2
7              increment wins[winner]
```

Output: the player with the most *wins*

Assume that steps 2 and 5–7 are each one elementary step. What is the asymptotic time complexity of this algorithm?

7.5.6. The nested loop in the previous algorithm actually generates far too many contests. Not only does it create situations in which the two players are the same person, necessitating the if statement, it also considers every head-to-head contest twice. The illustration on the left below (a) shows all of the contests created by the nested `for` loop in the previous function in a four-player tournament.



A line between two players represents the player on the left challenging the player on the right. Notice, for example, that Amelia challenges Caroline *and* Caroline challenges Amelia. Obviously, these are redundant, so we would like to avoid them. The red lines represent all of the unnecessary contests. The remaining contests that we need to consider are illustrated on the right side above (b).

(a) Think about how you can design an algorithm to create just the needed contests. Imagine that you are iterating from top to bottom over the *players* list on the left side of (b) above. Notice that each value of *player1* on the left only challenges values of *player2* on the right that come after it in the list. First, Amelia needs to challenge Beth, Caroline, and David. Then Beth only needs to challenge Caroline and David because Amelia challenged her in the previous round. Then Caroline only needs to challenge David because both Amelia and Beth already challenged her in previous rounds. Finally, when we get to David, everyone has already challenged him, so nothing more needs to be done.

Modify the nested for loop in the algorithm so that it implements this more efficient algorithm instead.

(b) What is the asymptotic time complexity of your algorithm? About how much faster is it than the original algorithm?

7.5.7. Suppose you have a list of projected daily stock prices, and you wish to find the best days to buy and sell the stock to maximize your profit. For example, if the list of daily stock prices was `[3, 2, 1, 5, 3, 9, 2]`, you would want to buy on day 2 and sell on day 5, for a profit of $8 per share. Similar to the previous exercise, you need to check all possible pairs of days, such that the sell day is after the buy day. Write a function

```
profit(prices)
```

that returns a tuple containing the most profitable buy and sell days for the given list of prices. For example, for the list of daily prices above, your function should return `(2, 5)`.

Selected Exercise Solutions

7.5.1 Removing the `if` statement does not affect the dictionary at all. If the item is already in the dictionary, its value is just assigned `True` again.

7.5.3
```
def duplicates(data):
    duplicated = { }
    duplicateItems = [ ]
    for item in data:
        if item not in duplicated:
            duplicated[item] = False
        elif not duplicated[item]:
            duplicated[item] = True
            duplicateItems.append(item)
    return duplicateItems
```