FIT1008 Introduction to Computer Science (FIT2085 for Engineers)

Tutorial 5

Semester 2, 2018

Exercise 1 *

Write a version of bubble sort that alternates left-to-right and right-to-left passes through the list. The left-to-right passes move the largest unserted element to its final place, the right-to-left passes move the smallest unsorted element to its place. This algorithm is called *shaker sort*.

(a) Give the best and worst case time complexity for your algorithm (together with the kind of list that will give that complexity), and explain why.

(b) Is this sorting method stable? Explain.

Solution

There are many ways to implement shaker short. One is as follows:

```
def swap(a_list, i, j):
    a_list[i], a_list[j] = a_list[j], a_list[i]

def shaker_sort(a_list):
    right = len(a_list)-1
    for left in range(right):
        for i in range(left, right):
             if a_list[i] > a_list[i+1]:
                  swap(a_list, i, i+1)
        right -= 1
        for i in range(right, left,-1):
             if a_list[i] < a_list[i-1]:
                  swap(a_list, i, i-1)
    return a_list
```

- The best time complexity for the algorithm is the same as the worst, since there is no way of stopping the loops early.
- The worst/best time complexity is $O(n^2)$, where n is the length of the list, because we have two nested loops both which depend linearly upon n, and whose inner operations are constant (we are still assuming the comparison is constant).
- The algorithm is stable since the relative order of equal valued items does not change (as the comparisons are both strict).

Note that

A better shaker short, which stops when the list is sorted, is as follows:

```
def better_shaker_sort(a_list):
    right = len(a_list)-1
    left = 0
    swapped = True
    left_to_right = True
    while left < right and swapped:
        swapped = False
        if left_to_right:
            for i in range(left, right):
                 if a_list[i] > a_list[i+1]:
                      swap(a_list, i, i+1)
                      swapped = True
            right -= 1
            left_to_right = False
        else:
            for i in range(right, left,-1):
                 if a_list[i] < a_list[i-1]:
                      swap(a_list, i, i-1)
```

```
                    swapped = True
            left += 1
            left_to_right = True
    return a_list
```

Note that the if-then-else inside the while loop avoids doing a left-to-right or right-to-left pass if `swapped` is still `False`.

For this algorithm the best time complexity is $O(n)$, where n is the length of the list, and it occurs when the list is sorted. This is because in this case, the algorithm stops after one left-to-right pass through the list. Since we are assuming the operations inside the inner loops are constant, we have $O(n)$. The worst time complexity is still $O(n^2)$, where n is the length of the list, and it occurs when the list is sorted in reverse order. The algorithm is also stable since the relative order of equal valued items does not change (as the comparisons are both strict).

And, in case you were wondering, the algorithm is incremental because if the list is already sorted and you put the new item at the start of the list, the bubble sort left-right will leave the new list sorted after one pass. What happens if you put it at the end? Think about it...

Note that `better_shaker_short` is more intricate than the `shaker_short` (more variables, if-then-elses, etc). We could also have done the following, where the only difference is the addition of swapped and the check of not swapped with a break.

```
def better_shaker_sort(a_list):
    right = len(a_list)-1
    for left in range(right):
        swapped = False
        for i in range(left, right):
            if a_list[i] > a_list[i+1]:
                swap(a_list, i, i+1)
                swapped = True
        right -= 1
        if not swapped:
            break
        for i in range(right, left,-1):
            if a_list[i] < a_list[i-1]:
                swap(a_list, i, i-1)
                swapped = True
        if not swapped:
            break
    return a_list
```

While the above one can be seen by some as easier to understand, the use of break is in general not recommended because it promotes unstructured code (i.e., it is easy to miss their effect on the flow). Further, it makes the life of the compiler more difficult, as it makes the logic of the loop less clear for a compiler or theorem prover.

Exercise 2 *

(a) Write a method for computing the sum of the digits of a number. For example, for number 979853562951413, the sum of its digits is $9+7+9+8+5+3+5+6+2+9+5+1+4+1+3 = 77$. To do this you can use integer division by 10 (//10) which returns an integer with the same digits except the last one, and reminder by 10 (% 10), which returns the last digit. For example, if you have X = 3456, then X//10 gives you 345, while X%10 gives you 6.

(b) Determine its complexity, in Big-O notation.

Solution

(a) One possible solution is as follows:

```
def sum_of_digits(x):
    sum = 0
    while x > 0:
        sum += x % 10
        x = x // 10
    return sum
```

(b): For complexity, observe that we start at the least significant digit of `x` (that is, 4 if `x` is 1234) and, for each position, we do a constant amount of processing, and that we move to the left one position at a time, shortening `x` as we go. So the time is $O(N)$, where N is the number of digits of `x`.

Note that, if the input to an algorithm is a number, then the complexity of the algorithm should be given in terms of the *number of digits* of the number, since that measures the size of the number *as input data to the algorithm*. Don't give the complexity in terms of the number itself. Doing so can be misleading. If the input is a number x, and the algorithm requires (say) x steps, then this might sound like linear time. But it is actually exponential time! If a number x has d decimal digits, then we know $10^{d-1} \leq x < 10^d$, so x is *exponential* in d.

This may take a little getting used to. But it's really not so surprising. Big numbers are easy to write down, but very hard to count up to! For example, if you write down 1 followed by say a hundred zeros — a "googol" (correctly spelt) — this might take you about a minute. But this number is many times larger than the number of atoms in the universe, and counting to it would take much longer than the current age of our universe.

By the way, it is not hard to give a precise formula for the number of digits d in a number x:

$$d = \lfloor \log_{10} x \rfloor + 1.$$

(Here, $\lfloor \cdots \rfloor$ means *truncate*.) If you want the number of digits used for representing the number x in some other base b, then just replace the base 10 for the logarithm here by b.

So the number of digits of x is $O(\log x)$.

Exercise 3 *

This question is designed to help you understand big-O complexity, and how it relates to the actual running time of a program.

Suppose you have a problem and you have designed five different algorithms to solve it, which you call Alpha, Bravo, Charlie, Delta and Echo. To study which is best, you implement them all and try them out. You run them on several inputs of each size $N = 10, 20, 30, \ldots, 90, 100$, and note down the longest running time, in milliseconds, for each input size. The data you collect is:

input size	maximum running time (ms)				
N	Alpha	Bravo	Charlie	Delta	Echo
10	9999999	10	20	9876543	1415926535
20	99999	40	200	3010	8979323846
30	220	777777	2000	4771	2643383279
40	330	160	20000	6021	5028841971
50	439	245	200000	6990	6939937510
60	549	360	2000000	7782	5820974944
70	660	489	20000000	8451	5923078164
80	767	640	200000000	9031	628620899
90	880	808	2000000000	9542	8628034825
100	989	1000	20000000000	10000	3421170679

Now, this data does not, of itself, *prove* anything about the time complexities of these algorithms. (Why not?) But it may still *suggest* to you some likely big-O expressions for the worst-case complexity of each algorithm.

(a) Give a plausible big-O expression for the worst-case time complexity of each of the algorithms Alpha, Bravo, Charlie, Delta and Echo. To do this, it might be useful to consider (as we did in lecture L10) what happens to the time when the input size N doubles or when it increases by any constant amount.

(b) Which algorithm is most scalable, according to big-O complexity? Which algorithm would you choose for an input of size 40? *Guess* which algorithm would you choose for an input of size 10000. (I'm not looking for detailed calculation for this input size, just wanting to get you thinking about it.) What about an input of size 100000000000000000000 (assuming you had a computer big enough to work with so much data!)?

Solution

The data itself does not *prove* anything about time complexity, because it does not give a mathematical proof about the behaviour of an algorithm for larger inputs (and time complexity is really *all about* what happens for large inputs). Also, the data does not necessarily cover *every* input of each size, so we cannot really be certain that we have the worst-case running time for each N. So all we can do is make plausible conjectures based on the patterns and trends we see in the data.

A good way to look at it is by looking at the time differences between the data. Lets look at the table in those terms (skipping the "extravagant" behaviour of some cells)

input size	maximum running time (ms)				
N	Alpha	Bravo	Charlie	Delta	Echo
10	9999999	10	20	9876543	1415926535
20	99999	40	200	3010	8979323846
30	220	777777	2000	4771	2643383279
40	330	160	20000	6021	5028841971
50	439	245	200000	6990	6939937510
60	549	360	2000000	7782	5820974944
70	660	489	20000000	8451	5923078164
80	767	640	200000000	9031	628620899
90	880	808	2000000000	9542	8628034825
100	989	1000	20000000000	10000	3421170679

(a) Time complexity of Alpha: apart from some initial extravagance, the running time seems well-behaved: in fact, as N increases by a constant, the time goes up by *at most* some constant amount. The increases in time are not perfectly regular, but that doesn't matter, as long as you can put a constant upper bound on them (in this case, it looks as is something close to 113 per every increase of 10). We can thus say that the running time is at most a constant times N. You can check the running time in the original table is in fact $\leq 10N$ (although by $N = 100$ the time is getting periliously close to 1000, so we might find out it is $\leq 11N$, rather than $\leq 10N$ if we measured time for number bigger than 100; to be even more precise, it is $\leq 11(N - 10)$). So, provided $N \geq 30$, the running time is $\leq 10N$. This means that the time complexity is $O(N)$.

Time complexity of Bravo: there is some kind of aberration at $N = 30$, but apart from this, it seems to grow more quickly than linear: as N goes up by constant steps, the time complexity goes up by *increasing* amounts (this is clear in the table above, but also in the original table, where by $N = 100$ it has taken over Alpha's time, even though for $N = 40$ it was smaller). But this does not narrow it down too much: there are lots of different functions that do this (e.g., $N \log N$, quadratic, cubic (or indeed any nonlinear polynomial, Exponential, ...). There are various ways of pinning the complexity down. For example, how does the time behave when the input size is doubled? In this case, it seems to go up by at most a factor of 4. This indicates $O(N^2)$. Further evidence comes from the thinly-disguised square numbers, multiplied by 10, in the running times.

Time complexity of Charlie: this increases by a *factor* of 10, every time you increase input size by a *constant* amount. This is a hallmark of *exponential* running time. But it is still important to find a value (or a good upper bound) for the *base* of the exponential, as it has a huge effect on running time. In this case, a little juggling with the numbers shows that the complexity is $O(10^{N/10})$.

Time complexity of Delta: again, we have an aberration for a particular small value of N, but hopefully we are learning that this does not matter for big-O complexity. So we look down the column and notice gradually increasing time, but that the time increases by decreasing amounts as N goes up by constant steps (see the table above). This tells us that the time complexity is sublinear, but we do not yet know more: maybe it could be $O(\sqrt{N})$, or $O(\sqrt[3]{N})$, or $O(\log N)$, or all sorts of other possibilties ... indeed, we might even wonder if it is bounded above by some fixed

constant, making it $O(1)$. One test is to see how the time behaves when the input size goes up by a constant *factor*. Suppose you double the input. If the time went up by a factor of at most $\sqrt{2}$, then the complexity would be $O(\sqrt{N})$. But that's too weak for the situation here. Suppose you start at $N = 20$ and then double to 40 and then 80. Notice that the time goes up by constant *amounts*, not by any fixed *factor*. This is a hallmark of logarithmic growth. So the complexity is $O(\log N)$.
Suppose, by the way, that you studied the behaviour of Delta for values of N going up by factors of 10. Then, in this case, the running times we obtained would be:

input size N	maximum running time (ms) Delta
10	9876543
100	10000
1000	20000
10000	30000
100000	40000
1000000	50000
10000000	60000
100000000	70000
1000000000	80000
10000000000	90000

The logarithmic behaviour of the time complexity is much clearer from this table.
Time complexity of Echo: these time values seem pretty random, some times increasing, some others decreasing (though in fact they are not; does anyone recognise them?). Although it would be hard to see any real pattern in these running times, remember that for big-O notation we are only looking for *upper bounds*. In this case we observe that all the running times are $\leq 10^{10}$. The data suggests no trend of growth (unlike for the other algorithms), so we conjecture that the running time is *always* $\leq 10^{10}$. In this case, the complexity is $O(1)$.

(b) Echo has the best big-O complexity, so it is the most scalable. But that does not mean it will always be the most efficient algorithm of choice for this problem; all it means is that it will *eventually* be the most efficient (i.e., for large enough inputs). For input size 40, it is clear from the table that Bravo is best, even though it has the second-worst big-O complexity of all the algorithms. For input size 10000, it turns out that Delta is best: its logarithmically-growing time complexity is still smaller than our upper bound (and the typical values) for the running time for Echo. But for the hypthetical input size of 10000000000000000000, the lowest big-O complexity will indicate the most efficient algorithm in this case: Echo.

Please note that the point of this exercise is not to figure out exact time functions such as $11(N-10)$, but for you to become familiar with some of the trademarks of time growth, such as:

- Linear if as N increases by a constant, time goes up by at most some constant.
- Quadratic if when N is doubled, time goes up by at most a factor of 4.
- Exponential on k if when N increases by a constant, time increases by a factor of k.
- Sublinear if when N increases by a constant, time increases by a decreasing amount.
- Constant if when N increases by a constant, time does not seem to have a growth pattern and is always below a certain constant, independent of N.

Exercise 4 *

What does the following snippet of code do? Discuss.

```
a = [0, 1]
try:
    b = a[2]
    print('that worked!')
except ValueError:
    print("no it didn't!")
```

Solution

The code shown doesn't print anything. Attempting to index past the length of a list throws an IndexError, which doesn't get caught by the `except ValueError` statement. Thus, the exception is not handled, and the program exits with a stack trace. In order to catch the error, we would have had to write the code thus:

```
a = [0, 1]
try:
    b = a[2]
    print('that worked!')
except IndexError:
    print("no it didn't!")
```

This code prints "no it didn't!" and exits cleanly, because the exception was caught and not re-raised.

Exercise 5

You can catch all exceptions by using a bare `except` statement, such as the one shown in the example below. However, this is in general acknowledged as a bad idea. Why do you think this is the case? And what would be the correct way to do it?

```
a = [0, 1]
try:
    b = a[2]
    c = int('foo')
    d = e
    f = 1/0
    print(1 + '1')
except:
    print("what happened!?")
```

Solution

Catching all exceptions with a blanket `except` statement forces you to handle *everything*.

- This is usually a bad idea, first because exception handling is supposed to be about the errors that you are expecting and, secondly, because not all errors are supposed to be handled by you.
- *Exception handling is about expected errors:* If you are opening a file, you brace for the file not being available, or the possibility of a hardware error. If you are sanitising user input, you know that users may enter values outside the scope of your function. Those are the errors you should catch. But if you implement a catch-all, you'll also supress the expression of many errors that you don't expect, and force the bugs to appear in other parts of your code.
- *Not all exceptions are supposed to be handled by you:* if you are writing library code, functions that other programmers will use in their own programs, there are three things you can do with exceptions that you receive: some of them you'll want to handle yourself. Some exceptions you'll want to catch, do something with them, and then raise them again so the calling code can handle them. And some of them you'll just let pass to the calling code. The blanket `except` makes more difficult for you to decide among these three choices.

Knowing this, and excusing the fact that the example is horribly contrived, how should we have written the code in the exercise? Here's a better option:

```
a = [0, 1]
try:
    b = a[2]
    c = int('foo')
    d = e
    f = 1/0
    print(1 + '1')
except IndexError:
    # handle the IndexError from a[2]
except ValueError:
```

```
    # handle the ValueError from int('foo')
except ZeroDivisionError:
    # handle the division by zero
except TypeError:
    # handle the Typeerror from "1 + '1'"
except Exception as e:
    # log exception
    raise e
```

Here the catch-all `except` is written in a modified form (`except Exception as e:`) that allows you to actually capture the exception and re-raise it. This is one of the few cases where capturing all exceptions is acceptable: you have already dealt with all errors you excepted, so you want to log anything you didn't expect before raising it in case it can be dealt with in the calling code.

Exercise 6

Definition: The *digital root* of a decimal integer is obtained by adding up its digits, and then doing the same to *that* number, and so on, until you get a single digit, which is the digital root of the number you started with.

For example, to find the digital root of 979853562951413, we calculate: sum of digits $= 9+7+9+8+5+3+5+6+2+9+5+1+4+1+3 = 77$, then sum of digits $= 7+7 = 14$, then sum of digits $= 1+4 = 5$. Now we have just one digit, 5, so that's the digital root of the number we started with.

(c) Write a method to compute the digital root of a positive integer.

(d) Determine its complexity, in Big-O notation.

Solution

(a): One possible solution is:

```
def digital_root(number):
    while number > 9:
        number = sum_of_digits(number)
    return number
```

(b): We know that, for a sufficiently large number x, the sum of the digits of x can be computed in time at most $K \cdot N$, where N is the number of digits of x and K is some constant. (This is because the complexity of `sum_of_digits` is $O(N)$, as explained in (b) above.) Now, the value returned by `sum_of_digits(x)` will be $\leq 9N$, since it is biggest when all the N digits of x are 9. How many digits does this value returned have? Certainly $\leq K' \cdot \log_{10}(9N)$, which is $K'(\log_{10} 9 + \log_{10} N)$ which in turn is $\leq K'' \log_{10} N$ (for appropriate constants K' and K''). The important point here is that the number of digits in `sum_of_digits(x)` is at most logarithmic in the number of digits in the original number x. This pattern continues, so we get a sequence of numbers whose numbers of digits are bounded above by constant multiples of N, $\log N$, $\log\log N$, $\log\log\log N$, ..., etc. The time taken for each `sum_of_digits` calculation is linear in the number of digits, but that number decreases very rapidly as we have just seen. So the total time will be at most some constant times $N + \log_N + \log\log N + \log\log\log N + \cdots$, which in turn is at most some bigger constant times N. (To see this, note that, for sufficiently large N, we have $\log N \leq N/2$, and $\log\log N \leq N/4$, etc (in fact these inequalities are very loose), so $N + \log N + \log\log N + \log\log\log N + \cdots \leq N + N/2 + N/4 + N/8 + \cdots \leq 2N$.)

So the total time taken by `digital_root` will be $O(N)$, where N is the number of digits in *number*. The constant (hidden by the Big-O notation) will be larger for `digital_root` than for `sum_of_digits`, though.

This discussion has been about worst-case complexity. (Remember that, if people just talk about "complexity" or "time complexity" or "computational complexity", without specifying best- or

worst-case, then, by default, it is taken to mean *worst-case* time complexity.) Best-case complexity of each algorithm is also $O(N)$. The hidden constant will be the same as for worst-case for `sum_of_digits`, since the procedure is the same whatever the digits are. However the hidden constant for `digital_root` will be a bit smaller, in the best case, than for the worst case, since in a number with a single non-zero digit followed by zeros, you only need to call `sum_of_digits` once, and then you get straight to the iteration for `digital_root`.

Other questions to consider are: How would you modify the above methods to work for all integers, not just positive ones? And how would you modify them to find digital roots in bases other than 10?

Extra challenges for the advanced student: Digital roots have some interesting properties. For example, an integer is divisible by 3 if and only if its digital root is 3, 6 or 9. An integer is divisible by 9 if and only if its digital root is 9. These observations can be useful for testing divisibility by pencil-and-paper (though they offer no advantage to a computer!). You don't get digital-root-based tests for numbers other than 3 or 9, though. You may care to try to prove that these divisibility tests for 3 and 9 work, and to see what divisibility tests you could do using digital roots for bases other than 10.