Algorithms and recursion

Read: *Chapter 13 from textbook*

- What programs implement is recipes for solving problems. These recipes are called **algorithms.**
- We have already seen (in the practice problems from a previous lab) an algorithm for searching a list of integers `nums` for a given integer `x`. The algorithm does the following:

```
1. Go over all elements in nums from the first to the last:
       if x==current element then return position of current element
2. Return -1
```

  The algorithm is called **linear search** (for obvious reasons) and returns either the first position where x is found (note that nobody told us that each list element appears only once), or -1 if x is not found. You can find the code in Section 13.1.2.
- If, in addition, we can guarantee that the list `nums` is *sorted from the smallest to the largest*, then we can do better (=faster). The algorithm can be described as follows:

```
1. Currently we search in the range [low..high] (Initially low=0, high=len(nums)-1)
2. If low>high then return -1
3. Let mid=(low+high)//2 be the position of the middle element in the searching range
4. If x==nums[mid] then
       return mid
   else if x>nums[mid] then
           low = mid+1
        else high = mid-1
5. Go to step 1.
```

  You can find the code for this algorithm in Section 13.1.3. It is called **binary search**. The basic idea is to break the searching range in three pieces: [low..mid-1], mid, [mid+1..high]. If x is not the mid element, then it must be either in the lower half [low..mid-1] (if x<nums[mid]) or the upper half (if x>nums[mid]); you can say this because your list is *sorted*. Now, it must be obvious where the advantage over linear search comes from: If we consider our most basic operation to be something like “pick up a list element and compare it with x”, then binary search continuously cuts the list in half, keeps the half that is relevant to x *and throws away the other half* (i.e., it will never examine elements in that half).

  Practice problem: Implement both searching methods and test them by searching for x=1, 99, 999, 9999, 99999, 999999, random() in a sorted list of numbers 0 – 1,000,000.
  Practice problem: To see that binary search works only when the list of numbers is sorted from the smallest to the largest, built an example list and an x where binary search outputs the wrong result (while linear search, of course, produces the correct result).

  From the first practice problem it should become clear that for x=999999 binary search is much faster than linear search. Why? Because linear search had to “touch” (pick-up and compare to x) 999999 elements, while binary search “touched” only one list element per halving. How many halvings were there? To go from 1,000,0000 elements to 1 (which is the worst that can happen to binary search, since it may get lucky and finish earlier) we need

$$((((1{,}000{,}000/2)/2)/2)\ldots)/2=1 \rightarrow 1{,}000{,}000*2^{-k}=1 \rightarrow k=\lceil \log 1{,}000{,}000 \rceil+1 \approx 25\, halvings$$

or only at most 25 list elements being examined (as compared to 999,999). Therefore, the **worst-case** for binary search in a sorted list with n elements is logn+1 basic operations, while for linear search it is n. If we use the worst-case as a measure of performance for algorithms, then binary search is much much better than linear search.

Practice problem: What would be a worst-case (max # of basic operations) input for binary search?

Recursion

- We have already encountered recursion before, in the recursive definition of functions such as factorial. If we view problems such as searching also as functions (as they indeed are; we write functions to implement such algorithms), in general the recursive definition of functions/problems has two main ingredients:
  1. A direct way to calculate/solve very small instances (called **base cases**).
  2. A way to calculate/solve an instance using the calculations/solutions of *strictly smaller* instances of exactly the same function/problem.

  We have already seen this, e.g., in the definition of Fibonacci sequences:
  1. fib(0)=1, fib(1)=1
  2. fib(n) = fib(n-1) + fib(n-2), n=2, 3, ...

  Note that we emphasized the *strictly smaller* instances used in the definition; this is important because it makes sense out of the definition: for example, we can calculate fib(n) by calculating f(0), f(1), f(2),...,f(n-2), f(n-1) in this order.

  Functions like Fibonacci or Factorial have straight-forward recursive definitions. But what about problems like searching a list? The idea is still the same: split the problem instance into strictly smaller instances of exactly the same problem, until you reach base cases for which you know the solution directly. For the linear search algorithm, we can reduce the given instance (x, list) into solving a instance (x, list') with a list' that is strictly smaller than list. How? The basic idea is that linear search can be broken into 2 parts: first check whether x is the first element of the list list[0]; if it is, then return 0, else return the result of linear search for x in the list list[1:] (the original list minus the first element that we have already checked). There are **two** base cases for which we know the answer: first, the case of the empty list [] (return -1), and, second, the case of x=list[0] (return 0).

```
def linear_search(x, nums):
    if nums==[]:                    #  base case: nums is empty
        return -1
    elif x==nums[0]:           # x is the first element of list nums
        return 0
    else:
        return linear_search(x, nums[1:])     # linear-search the rest of the list
```

You may have already seen the problem with this recursive algorithm: while it will return the right answer (-1) if x is not in the list, it always returns 0 as the position if x is in the list! To see this, try to run the code by hand for, say, x=3 and nums=[1,2,3,4]. Where exactly is the problem? Well, notice that the way we shrink the problem is the reason! Here is the running of the algorithm in the suggested example:

Initial call:

Current instance	Returned (i.e., calculated) value
linear-search(3, [1,2,3,4])	?

First recursive call:

Current instance	Returned (i.e., calculated) value
linear-search(3, [1,2,3,4])	?
linear-search(3, [2,3,4])	?

Second recursive call:

Current instance	Returned (i.e., calculated) value
linear-search(3, [1,2,3,4])	?
linear-search(3, [2,3,4])	?
linear-search(3, [3,4])	0

After the second recursive call returns:

Current instance	Returned (i.e., calculated) value
linear-search(3, [1,2,3,4])	?
linear-search(3, [2,3,4])	0

After the first recursive call returns:

Current instance	Returned (i.e., calculated) value
linear-search(3, [1,2,3,4])	0

Finally, the original call returns 0.

The root of the problem is clearly the second recursive call, that (correctly) identifies x=3 in the current list=[3,4], but (incorrectly) sees x as the first element of that list (position 0). There are at least two ways to remedy this mistake: a straight-forward one and a somewhat more involved one.

Practice problem: Change the algorithm to check whether x is equal to the *last* element of nums instead of the first; change the answer of the second base case.
Practice problem: Add one more parameter to the function linear_search, that keeps track of how many elements of the original list have been dropped so far. So the definition should look like:

```
def linear_search(x, nums, drop)
```

and initially you call it with drop=0. Now you know what to return (instead of 0) when x is found.

- Take a better look at the consecutive calls of linear_search as described in the table above. We record every new call right after the last call (which is the call that induced the new call), until the last call doesn't make any more recursive calls because it has reached a base case; then it returns the answer to its caller and is removed from the table. Therefore, in order to keep track of the execution of a recursive algorithm, all we need is to keep a table: (i) We **add** *at its end* every new call we make, after we have recorded some information for the caller to be able to continue its execution after the new call returns (such as the current line of code, the current values of parameters and variables). (ii) We **remove** the *last call* in the table when it returns (note that it is the last call that will be the first to return!) The table and its evolution looks like an upside-down stack of "plates", where new "plates"/calls are added to or removed from its top. Hence, such a scheme is called, in general, a **stack** and the addition/removal operations are called **push** and **pop** respectively.
- Now it should be clear that base cases end up being the last "plate" pushed to the top of the stack, and after them we can start popping "plates" by returning calculated values. If we miss one or more base cases, we run the danger of continuously pushing "plates" to the stack without ever popping anything, until we either crush by running out of memory or some other error occurs (e.g., running out of bounds)! To see this, rewrite the code for linear_search but leaving out the case nums==[]. Then run it for (x=3, nums=[1,2,3,4]) and for (x=33, [1,2,3,4]). The first case should work as expected, but the second produces an "out of bounds" error! See Section 13.2.3 for an example of running out of memory because of *infinite recursion*.

Practice problem: Figure 13.1 shows the sequence of calls for the recursive implementation of Factorial in Section 13.2.2. Follow the figure step-by-step building a table like the one above along the way. (By the way, in the first call, the figure shows n as having no value; that is wrong: n=5 for the initial call.)
Practice problem: To see how the recursive calls are "stacked", write a recursive algorithm that takes a number in base 10 and prints the same number in base 8. Recall that the mathematical procedure is to start with the number n, n%8 is the least significant digit in base 8, and continue with the conversion of the number n//8 to base 8, until the number to convert becomes 0 (then you stop). So your recursive function probably would look like:

```
1. if n==0:
2.       return;
3. digit = n%8
4. convert(n//8)
5. return
```

The question is where exactly do you put the print(digit)? Between lines 3-4 or between lines 4-5? Try to think about this (i.e., build the tables...) before you go ahead and try it in the code.

- Binary search was also described as a recursive algorithm (although we implemented it as a loop). It is actually described in more detail in p. 432. Note that the recursive implementation passes low and high as parameters, for exactly the same reason we had to pass drop in linear_search (verify this!); the Python code can be found in Section 13.2.6.

  Practice problem: Compare the running times of the iterative and recursive implementations of linear_search with a huge list (say, 1,000,000 elements) and an x around the middle. You should see a big deference, with the recursive implementation being slower; obviously, all this stack book-keeping takes time. More details can be found in Section 13.2.7.
  Practice problem: There is a recursive algorithm for fast exponentiation in Section 13.2.5. Change the definition so that instead of dividing the exponent n by 2, we divide it by 3. *(Hint: The number of base cases is equal to the remainders in the division of the exponent n: if the exponent is divided by 2, then there are two possible remainders n%2 = 0 ,1; if divided by 3, then there are three possible remainders n%3 = 0, 1, 2.)*

- Searching is a very basic computational problem. An equally basic problem is that of **sorting**. Sorting can be applied to a list of items for which we have an ordering, i.e., a way of comparing two items. In Section 13.3 there is the description (and implementation) of two well-known sorting algorithms: selection-sort and merge-sort.

  Practice problem: What is the worst-case unsorted list for selection-sort? What is the worst-case list for merge-sort? (worst-case means a list that forces the algorithm to do the most possible work)
  Practice problem: Implement both algorithms and try them for lists with 10, 1000, 100000 numbers *in reverse order*. Try the same with lists constructed with random numbers (using random(), for example). Are your running times consistent with Figure 13.4? (Read Section 13.3.3 to make sense of this.)
  Practice problem: From the previous problem and Section 13.3.3 one starts realizing that selection-sort needs about n^2 steps in the worst case and merge-sort needs about $n log_2 n$ steps, and that's why merge-sort is faster once n becomes big enough. But these worst-case running times are peanuts compared to the 2^n steps needed for the Towers of Hanoi problem in Section 13.4.1. Implement the algorithm, to see how huge the running time becomes even for small n. And yet, even *that* is peanuts compared to the Halting Problem described in Section 13.4.2: this problem has *no algorithm!!!* So we have seen worst-case running times whose dependence on the size of the input n is *polynomial* (selection-sort, merge-sort), *exponential* (Towers of Hanoi), or the problem may be *undecidable* (Halting Problem).
  Practice problem: Do all textbook problems (except the ones related to graphics).