Elementary

1. Write a program that prints 'Hello World' to the screen.
2. Write a program that asks the user for their name and greets them with their name.
3. Modify the previous program such that only the users Alice and Bob are greeted with their names.
4. Write a program that asks the user for a number `n` and prints the sum of the numbers 1 to `n`
5. Modify the previous program such that only multiples of three or five are considered in the sum, e.g. 3, 5, 6, 9, 10, 12, 15 for `n`=17
6. Write a program that asks the user for a number `n` and gives them the possibility to choose between computing the sum and computing the product of 1,…,`n`.
7. Write a program that prints a multiplication table for numbers up to 12.
8. Write a program that prints *all* prime numbers. (Note: if your programming language does not support arbitrary size numbers, printing all primes up to the largest number you can easily represent is fine too.)
9. Write a guessing game where the user has to guess a secret number. After every guess the program tells the user whether their number was too large or too small. At the end the number of tries needed should be printed. It counts only as one try if they input the same number multiple times consecutively.
10. Write a program that prints the next 20 leap years.
11. Write a program that computes

$$4 \cdot \sum_{k=1}^{10^6} \frac{(-1)^{k+1}}{2k-1} = 4 \cdot (1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 \ldots).$$

Lists, Strings

1. Write a function that returns the largest element in a list.
2. Write function that reverses a list, preferably in place.
3. Write a function that checks whether an element occurs in a list.
4. Write a function that returns the elements on odd positions in a list.
5. Write a function that computes the running total of a list.
6. Write a function that tests whether a string is a palindrome.
7. Write three functions that compute the sum of the numbers in a list: using a `for`-loop, a `while`-loop and recursion. (Subject to availability of these constructs in your language of choice.)
8. Write a function `on_all` that applies a function to every element of a list. Use it to print the first twenty perfect squares. The perfect squares can be found by multiplying each natural number with itself. The first few perfect squares are `1*1= 1`, `2*2=4`, `3*3=9`, `4*4=16`. Twelve for example is not a perfect square because there is no natural number

`m` so that `m*m=12`. (This question is tricky if your programming language makes it difficult to pass functions as arguments.)

9. Write a function that concatenates two lists. `[a,b,c]`, `[1,2,3]` → `[a,b,c,1,2,3]`
10. Write a function that combines two lists by alternatingly taking elements, e.g. `[a,b,c]`, `[1,2,3]` → `[a,1,b,2,c,3]`.
11. Write a function that merges two sorted lists into a new sorted list. `[1,4,6]`,`[2,3,5]` → `[1,2,3,4,5,6]`. You can do this quicker than concatenating them followed by a sort.
12. Write a function that rotates a list by `k` elements. For example `[1,2,3,4,5,6]` rotated by two becomes `[3,4,5,6,1,2]`. Try solving this without creating a copy of the list. How many swap or move operations do you need?
13. Write a function that computes the list of the first 100 Fibonacci numbers. The first two Fibonacci numbers are 1 and 1. The `n+1`-st Fibonacci number can be computed by adding the `n`-th and the `n-1`-th Fibonacci number. The first few are therefore 1, 1, 1+1=2, 1+2=3, 2+3=5, 3+5=8.
14. Write a function that takes a number and returns a list of its digits. So for `2342` it should return `[2,3,4,2]`.
15. Write functions that add, subtract, and multiply two numbers in their digit-list representation (and return a new digit list). If you're ambitious you can implement Karatsuba multiplication. Try different bases. What is the best base if you care about speed? If you couldn't completely solve the prime number exercise above due to the lack of large numbers in your language, you can now use your own library for this task.
16. Write a function that takes a list of numbers, a starting base `b1` and a target base `b2` and interprets the list as a number in base `b1` and converts it into a number in base `b2` (in the form of a list-of-digits). So for example `[2,1,0]` in base 3 gets converted to base 10 as `[2,1]`.
17. Implement the following sorting algorithms: Selection sort, Insertion sort, Merge sort, Quick sort, Stooge Sort. Check Wikipedia for descriptions.
18. Implement binary search.
19. Write a function that takes a list of strings an prints them, one per line, in a rectangular frame. For example the list `["Hello", "World", "in", "a", "frame"]` gets printed as:

```
*********

* Hello *
* World *
* in    *
* a     *
* frame *
*********
```

26. Write function that translates a text to Pig Latin and back. English is translated to Pig Latin by taking the first letter of every word, moving it to the end of the word and adding 'ay'. "The quick brown fox" becomes "Hetay uickqay rownbay oxfay".

Intermediate

1. Write a program that outputs all possibilities to put + or – or nothing between the numbers 1,2,…,9 (in this order) such that the result is 100. For example 1 + 2 + 3 - 4 + 5 + 6 + 78 + 9 = 100.
2. Write a program that takes the duration of a year (in fractional days) for an imaginary planet as an input and produces a leap-year rule that minimizes the difference to the planet's solar year.
3. Implement a data structure for graphs that allows modification (insertion, deletion). It should be possible to store values at edges and nodes. It might be easiest to use a dictionary of (node, edgelist) to do this.
4. Write a function that generates a DOT representation of a graph.
5. Write a program that automatically generates essays for you.
    1. Using a sample text, create a directed (multi-)graph where the words of a text are nodes and there is a directed edge between `u` and `v` if `u` is followed by `v` in your sample text. Multiple occurrences lead to multiple edges.
    2. Do a random walk on this graph: Starting from an arbitrary node choose a random successor. If no successor exists, choose another random node.
6. Write a program that automatically converts English text to Morse code and vice versa.
7. Write a program that finds the longest palindromic substring of a given string. Try to be as efficient as possible!
8. Think of a good interface for a list. What operations do you typically need? You might want to investigate the list interface in your language and in some other popular languages for inspiration.
9. Implement your list interface using a fixed chunk of memory, say an array of size 100. If the user wants to add more stuff to your list than fits in your memory you should produce some kind of error, for example you can throw an exception if your language supports that.
10. Improve your previous implementation such that an arbitrary number of elements can be stored in your list. You can for example allocate bigger and bigger chunks of memory as your list grows, copy the old elements over and release the old storage. You should probably also release this memory eventually if your list shrinks enough not to need it anymore. Think about how much bigger the new chunk of memory should be so that your performance won't be killed by allocations. Increasing the size by 1 element for example is a bad idea.
11. If you chose your growth right in the previous problem, you typically won't allocate very often. However, adding to a big list sometimes consumes considerable time. That might be problematic in some applications. Instead try allocating new chunks of memory for new items. So when your list is full and the user wants to add something, allocate a new chunk of 100 elements instead of copying all elements over to a new large chunk. Think about where to do the book-keeping about which chunks you have. Different book keeping strategies can quite dramatically change the performance characteristics of your list.
12. Implement a binary heap. Once using a list as the base data structure and once by implementing a pointer-linked binary tree. Use it for implementing heap-sort.

13. Implement an unbalanced binary search tree.
14. Implement a balanced binary search tree of your choice. I like (a,b)-trees best.
15. Compare the performance of insertion, deletion and search on your unbalanced search tree with your balanced search tree and a sorted list. Think about good input sequences. If you implemented an (a,b)-tree, think about good values of a and b.

Advanced

1. Given two strings, write a program that efficiently finds the longest common subsequence.
2. Given an array with numbers, write a program that efficiently answers queries of the form: "Which is the nearest larger value for the number at position `i`?", where distance is the difference in array indices. For example in the array `[1,4,3,2,5,7]`, the nearest larger value for 4 is 5. After linear time preprocessing you should be able to answer queries in constant time.
3. Given two strings, write a program that outputs the shortest sequence of character insertions and deletions that turn one string into the other.
4. Write a function that multiplies two matrices together. Make it as efficient as you can and compare the performance to a polished linear algebra library for your language. You might want to read about Strassen's algorithm and the effects CPU caches have. Try out different matrix layouts and see what happens.
5. Implement a van Emde Boas tree. Compare it with your previous search tree implementations.
6. Given a set of d-dimensional rectangular boxes, write a program that computes the volume of their union. Start with 2D and work your way up.