Designing Algorithms

What is algorithm: An **algorithm** is a procedure or formula for solving a problem. The word derives from the name of the mathematician, Mohammed ibn-Musa al-Khwarizmi, who was part of the royal court in Baghdad and who lived from about 780 to 850.

Algorithm is step-by-step set of operations to be performed.

Algorithms perform:

- Calculation,
- Data processing, and/or
- Automated tasks.

Simple: Example

Designing Algorithms

In this class, you'll learn an algorithm-writing technique called:

top-down design.

You start by describing your solution in English and then mark the phrases that correspond directly to Python statements.

Those that don't correspond are then rewritten in more detail in English, until everything in your description can be written in Python.

Searching for the Smallest Values

Problem case: find the index of the smallest items in an unsorted list

The sample data number of humpback whales seen on the coast of British Columbia over the past ten years:

809	834	477	478	307	122	96	102	324	476

We'll go through three top down design to solve this problem:

The algorithm for this problem is simple:

- Apply the minimum method for the list
- Find the index of the minimum

```
>>> counts = [809, 834, 477, 478, 307, 122, 96, 102, 324, 476]
>>> low = min(counts)
>>> min_index = counts.index(low)
>>> print(min_index)
```

Summer code:

```
>>> counts = [809, 834, 477, 478, 307, 122, 96, 102, 324, 476]
>>> counts.index(min(counts))
```

find the indices of the *two* smallest values'

Here are three high-level description.
Each is the first step in doing a top-down design solution.

Alg1:

1) *Find, remove, find.* Find the index of the minimum, remove that item from the list, and find the index of the new minimum item in the list. Then we find the second index.

2) *Sort, identify minimums, get indices.*

3) *Walk through the list.* Examine each value in the list in order, keep track of the two smallest values found so far, find their indices.

find the indices of the *two* smallest values

While you are investigating these algorithms , consider this question:
Which one is the fastest?

Alg1- step1 of the top down design:

We simplify the problem by means of English statement comes as comment

English statements comments, makes it easier to understand *why* the Python code does what it does:

find the indices of the *two* smallest values'

Step2: Next step in the top down design we have to translate some of the comment into Python code:

In this step we start with the most obvious and clear comment, start translating these comment into Python code:

The top statements;

Find smallest
Find index of smallest
Remove smallest

Find next smallest
Find index of the next

find the indices of the *two* smallest values

Step3:

- Since we removed the smallest item,
- we need to put it back.
- Also, when we remove a value,
- the indices of the following values shift down by one.
- So, since smallest has been removed,
- That is mean the index moved back one spot
- if we want to get the indices of the two lowest values in the *original* list,
- we might need to add 1 to min2:

find the indices of the *two* smallest values'

That seems like a lot of work, and it is. Even if you go right to code, you'll be thinking through all those steps. But by writing them down first, you have a much greater chance of getting it right with a minimum amount of work.

recall:

Remember that detailing the problem and writing the problem specification are vital to the successful implementation of any potential automated business/scientific process. In practice, arriving at a well-defined problem specification is extremely important, and very much possible that automated business project or processes have failed in implementation due to an inadequate problem specification.

Alg2:

Sort, Identify Minimums, Get Indices

Alg2-Step 1 of the top down design:

```
def find_two_smallest(L):

""" (list of float) -> tuple of (int, int)
Return a tuple of the indices of the two smallest values in list L.
>>> find_two_smallest([809, 834, 477, 478, 307, 122, 96, 102, 324, 476])
(6, 7)
"""
    # Sort a copy of L
    # Get the two smallest numbers
    # Find their indices in the original list L
    # Return the two indices
```

Alg2:

Sort, Identify Minimums, Get Indices

Alg2-Step 2 of the top down design:

```
# Get a sorted copy of the list so that the two smallest items
# are at the  front
```

Alg2-Step 3 of the top down design:

```
# Find their indices in the original list L
# Return the two indices
```

Alg3

Walk Through the List

Alg3-step1:

```
def find_two_smallest(L):
""" (list of float) -> tuple of (int, int)
Return a tuple of the indices of the two smallest values in list L.
>>> find_two_smallest([809, 834, 477, 478, 307, 122, 96, 102, 324,
476])
(6, 7)
"""
    # Examine each value in the list in order
    # Keep track of the indices of the two smallest values found so far
    # Update these values when a new smaller value is found
    # Return the two indices
```

Alg3

Walk Through the List

Alg3-step2:

- We have to create loop to examine all items in the list
- Every loop has three parts:
  - An initialization section to set up the variables
  - A loop condition, and
  - A loop body.
- The initialization will set up min1 and min2, which will be the indices of the smallest two items encountered so far.
- min1 and min2 are set to the first two items of the list
- We have set min1 to the smallest among the first two items

Alg3

Walk Through the List

Alg3-step3:

# Examine each value in the list in order
# Update these values when a new smaller value is found
# Return the two indices
In this step we have couple of choices:

- We can iterate with a for loop over the values,
- for loop over the indices, or
- a while loop over the indices.
- Since we're trying to find indices, we'll use a for loop over the indices
- we'll start at index 2 because we've examined the first two values
- At the same time, we'll mention min1 and min2 in side the loop body.

Alg3

Walk Through the List

Alg3-step3 The anatomy of the if statement :

```
# L[i] is smaller than both min1 and min2, in between, or
#  larger than both:

    #  If L[i] is smaller than min1 and min2, update them both

    #  If L[i] is in between, update min2

    #  If L[i] is larger than both min1 and min2, skip it
```

Timing the Functions

The main two crucial factors of SW Quality assurance are:

- *Timing :* how long it takes to run
- *Memory:* how much memory it uses

Time and space—are fundamental to the theoretical study of algorithms.

Fast programs are more useful than slow ones, and
programs that need more memory than what your computer has
aren't particularly useful at all.

Timing the Functions

Module time contains functions related to time.
One of these functions is:

perf_counter, which returns a time in seconds.

We can call it before and after the code we want to time and take the difference to find out how many seconds have elapsed.

We multiply by 1000 in order to convert from seconds to milliseconds:

EX: fid_time()

Timing the Functions

Next: find execution time for the three find_two_smallest functions.

- Rather than run the time individually for the three functions

- Create a function that takes another function as a parameter as well as the list to search in.

- This timing function will return how many milliseconds it takes to execute each function.

- Timing function as main program, reads the file of sea level pressures and each of the find_two_smallest functions as parameters

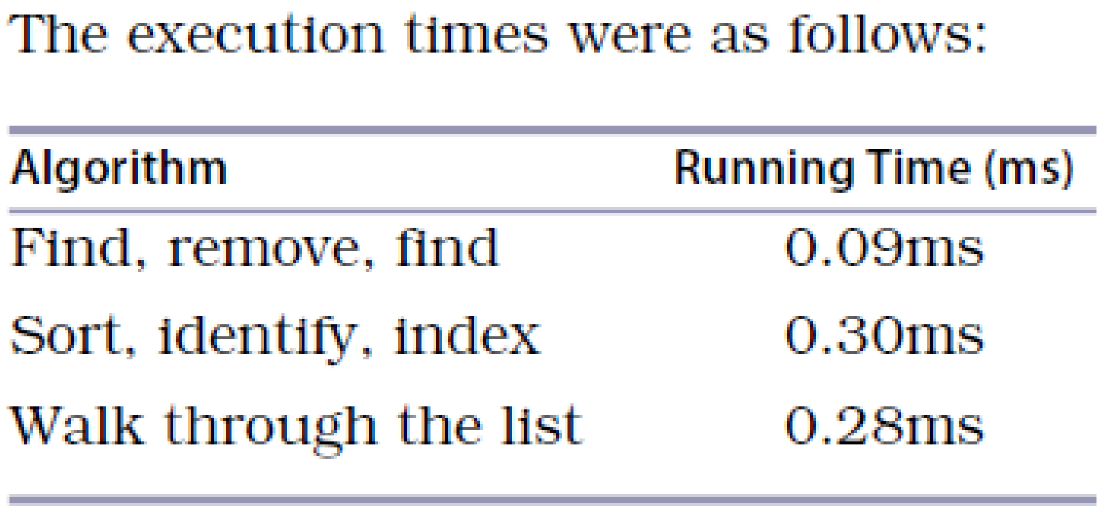
Timing the Functions

Algorithm for find timing for many functions:

1. *create function whose parameters is the called function and the list*
2. *Set the initial time and the final time of execution*
3. *Return how many millisecond each function needs to run*
4. *Prepare the data to be read by each function*
5. *Pass the function and the data to the main function*
6. *Execute each of the called function*
7. *Print the time of execution for each*

EX:

Timing the Functions



The execution times were as follows:

Algorithm	Running Time (ms)
Find, remove, find	0.09ms
Sort, identify, index	0.30ms
Walk through the list	0.28ms

- Time is mall, no human being can notice the difference
- It is trade off between time of execution and clarity in the code
- Process list with not very big amount of data, we prefer simplicity or clarity rather than speed.
- Process millions of values? Then we might choose better execution time

if **if** __name__ == *'__main__'*:
Used to check if python run directly then the name = main

When we imported module then the name of the running code is the imported one
To check if the file run directly or imported.

You can print this statement after the if to see which module is executed:

 #print('module is : ', __name__)