

# Theory of Algorithms

ian.mcloughlin@gmit.ie

# Topics

Python

Permutations

Timing Algorithms

Functional Programming

Turing Machines

Complexity Classes

# Python

# About Python

**January 1994** – Python 1.0.0 released.

**Guido van Rossum** – Designer/Author of Python.

**Current versions** – 3.5.1 and 2.7.11.

**Interpreted** – Python implementation must be present at runtime.

**Off-side rule** – Blocks identified by indentation, as opposed to curly braces.

**Popularity** – IEEE Spectrum ranks it as the fourth most popular language (July 2015).

**Community** – Python Enhancement Proposals, notably PEP 8: The Python Style Guide.



- Started Python as a hobby.
- Worked for Google, half-time spent on Python.
- Now works at Dropbox.
- Benevolent dictator for life (BDFL).

---

```
1 x = int(raw_input("Please enter an integer: "))
2 if x < 0:
3     x = 0
4     print 'Negative changed to zero'
5 elif x == 0:
6     print 'Zero'
7 elif x == 1:
8     print 'Single'
9 else:
10    print 'More'
```

---

# Loops

---

```
1 # A for loop.  
2 a = ['Mary', 'had', 'a', 'little', 'lamb']  
3 for i in range(len(a)):  
4     print(i, a[i])
```

---

---

```
1 # A while loop.  
2 a, b = 0, 1  
3 while b < 1000:  
4     print(b)  
5     a, b = b, a+b
```

---

*[docs.python.org/3/tutorial](https://docs.python.org/3/tutorial)*

---

```
1 # write Fibonacci series up to n
2 def fib(n):
3     """Print a Fibonacci series up to n."""
4     a, b = 0, 1
5     while a < n:
6         print(a)
7         a, b = b, a+b
```

---



**Reference implementation** – Many different Python implementations exist.

**Version 3** – Broke backwards compatibility (somewhat).

**Unladen Swallow** – Google attempt to fix some Python problems.

**Modules** – Lots of great Python modules available.

**Lists** in Python are usually written as comma-separated values between square brackets.

**Types** – elements of a list don't have to have the same types.

**Slicing** is possible, where we take a sublist of the list.

**Assignment** to slices is possible.

**len()** is a built-in function that returns the length of a list.

**range()** is a built-in function that returns a list of numbers.

Note: it returns an *iterator*.

---

```
1 letters = ['a', 'b', 'c']
2 letters[1:] = ['c', 'd']
3 range(10) # [0,1,2,3,4,5,6,7,8,9]
```

---

**Strings** are a lot like lists in Python.

**Assignment** to slices is not allowed, however.

---

```
1 words = "This is a sentence."
2 words[8]          # a
3 words[5:7]        # is
4 words[:7]         # This is
5 words[10:]        # sentence.
6 words[17:9:-1]    # ecnetnes
7
8 len(words)        # 19
9 "One" + "Two"     # OneTwo
```

---

**def** is the keyword for defining a function.

**Parameters** can be given defaults, so that they are optional.

---

```
1 def axn(x, a=1, n=2):
2     return a*(x**n)      #  $ax^n$ 
3
4 axn(3)                    # 9
5 axn(3, 2)                # 18
6 axn(3, 2, 3)             # 54
7 axn(3, n=3)              # 27
```

---

**Comprehensions** are quick ways of creating lists from other lists.

---

```
1 nos = range(5) # [0, 1, 2, 3, 4]
2 squares = [i*i for i in nos] # [0, 1, 4, 9, 16]
3 oddsqs = [i*i for i in nos if i % 2 == 1] # [1, 9]
```

---

**map()** takes a function and a list.

**New list** – it returns a new generator, which is the original list with the function applied to each element.

---

```
1 map(len, words)
2 list(map(len, words))
```

---

**lambda** functions are short, inline functions.

**Nameless** – lambda functions need not have a name.

---

```
1 lambda x: x + n
```

---

# Without generators

---

```
1 # Build and return a list
2 def firstn(n):
3     num, nums = 0, []
4     while num < n:
5         nums.append(num)
6         num += 1
7     return nums
8
9 sum_of_first_n = sum(firstn(1000000))
```

---



---

```
1 # yields items instead of returning a list
2 def firstn(n):
3     num = 0
4     while num < n:
5         yield num
6         num += 1
7
8 sum_of_first_n = sum(firstn(1000000))
```

---

# Permutations

**Permutations** are rearrangements of ordered collections of items.

**Example:** “abcd” is a rearrangement of “bacd”.

**Think** of having a four boxes, where we have to place one of the items in each box.

**What** are all the different ways of doing this?

**What** are all the different ways of associating items with boxes?

We can consider permutations in abstraction. For instance, if we have four items to rearrange, we can label the first item 1, the second 2, and so on. Then we can represent the various permutations, in terms of the numbers associated with the items. The permutations “abcd” and “bacd” could be represented by:

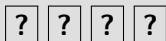
$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 3 & 4 \end{pmatrix}$$

In this way we can consider permutations in their own right.

# Counting permutations

With four items, how many distinct permutations are there?

Consider having four placeholders where we can place the items:



When we place an item in the first box, we have four choices, then we are left with only three choices for the second, two choices for the third, and one choice (i.e. not a choice at all) for the last.

So for there are  $4 \times 3 \times 2 \times 1 = 4!$  choices in total.

# Counting anagrams

## Repetition

What happens when we consider two of our items to be the same? For instance, what if we are looking for all distinct rearrangements of “aacd” as opposed to “abcd”? In that case we need to account for the rearrangements of those items by dividing by the factorial of the number of times each item is repeated:  $\frac{4!}{2!}$ .

If more than one item is repeated a number of times we just keep dividing by the factorials of the numbers of repetitions. The distinct number of rearrangements of “aaabbcd” is  $\frac{7!}{3!2!}$ .

## Exercise

Calculate the number of distinct rearrangements of the word “Mississippi”.

# Heap's algorithm

**Heap** published an algorithm in November 1963 for generating permutations.

**Published** in The Computer Journal.

**Read** the article in the link below – it's an easy read.

**Pairs** of items are interchanged to generate each new permutation.

**Induction** is used to show the algorithm works.

# Heap's algorithm description

- Suppose we know how to permute  $(n - 1)$  items.
- That is, we know all of the different ways of slotting  $(n - 1)$  items in  $(n - 1)$  boxes.
- Let's add another,  $n^{th}$  item, and another box, an  $n^{th}$  box.
- First, place the  $n^{th}$  item in the  $n^{th}$  box.
- Permute all the the other items, which you know how to do.
- Then swap another item with the  $n^{th}$  item.
- Again permute the items in boxes 1 to  $(n - 1)$ .
- Swap the item in the  $n^{th}$  box with another, different item.
- Repeat until all items have been in box  $n$ .



# Steinhaus-Johnson-Trotter algorithm

**Johnson** published another algorithm in 1963 for generating permutations.

**Attributed** to three people: Steinhaus, Johnson and Trotter.

**See** the article in the link below – it's a trickier read.

**Pairwise** – the algorithm can be done pair-wise, like Heap's.

**Induction** is used to show the algorithm works.

# Steinhaus-Johnson-Trotter algorithm description

- Start with two items, 1 and 2, and generate their list of permutations 12 and 21.
- Use the two-item list to generate the three item list in the following way:
  - Place 3 at the right of the first element in the two-item list.
  - Then move 3 one place to the left continuously until its on the left.
  - Then place 3 at the left of the next element in the two-item list.
  - move 3 one place to the right continuously until it reaches the right.
- Use the three item list to generate the four item list in the same way, and so on.

---

```
1  for permutation in permutations(letters):  
2      checkIfWord(permutation)
```

---

---

```
1 worddict = {}  
2  
3 for word in dictionaryOfWords:  
4     sortword = sorted(list(word))  
5     hashword = hash(sortword)  
6     allWords = worddict.get(hashword, set())  
7     allWords.update({word})  
8     worddict[hashword] = allWords
```

---

## Conundrum – Quick method checking

---

```
1 word = "conundrum"
2 sortword = sorted(list(word))
3 hashword = hash(sortword)
4
5 worddict.get(hashword, None)
```

---

# Timing Algorithms

---

```
1 $ python -m timeit '"-".join(str(n) for n in range(100))'
2 10000 loops, best of 3: 30.2 usec per loop
3 $ python -m timeit '"-".join([str(n) for n in range(100)])'
4 10000 loops, best of 3: 27.5 usec per loop
5 $ python -m timeit '"-".join(map(str, range(100)))'
6 10000 loops, best of 3: 23.2 usec per loop
```

---

---

```
1  import timeit
2
3  def test():
4      """Stupid test function"""
5      L = [i for i in range(100)]
6
7  if __name__ == '__main__':
8      import timeit
9      print(timeit.timeit("test()",
10                          setup="from __main__ import test"))
```

---



# Functional Programming



- John McCarthy while at MIT – late 1950s.
- Created Lisp.
- Lisp generally considered first functional programming language (not really though).
- Lots of dialects exist today, such as Scheme and Common Lisp.

# For loops

---

```
1 inds = list(range(10))
2 total = 0
3 for i in inds:
4     total = total + (i * 7)
```

---

---

```
1 def byseven(i):
2     return i * 7
3
4 inds = list(range(10))
5 sevens = map(byseven, inds)
6 total = sum(sevens)
```

---

**Imperative** programming is a programming paradigm where statements are used to change the *state*.

**State** is the name given to the current data/values related to an executing process, including internal stuff like the call stack.

**Processes** begin with an initial state and (possibly) have (a number of) halt states.

**Statements** change the state.

**Functions** in imperative programming languages might return different values for the same input at different times, because of the state.

**Functional** programming languages (try to) not depend on state.

**Functions** are said to have side effects if they modify the state (on top of returning a value).

**Static and global** variables are often good examples of side effects in action.

**Functional** programming tries to avoid side effects.

**It's tricky** to avoid them – such as when we need user input.

# Turing Machines

# Complexity Classes