linear Relating to a straight line. Here, we talk about graphing how the time taken by an algorithm depends on the size of the data it is processing. Linear algorithms have straight-line graphs that can describe this relationship.

linear search A search that probes each item in a list or sequence, from first, until it finds what it is looking for. It is used for searching for a target in unordered lists of items.

Merge algorithm An efficient algorithm that merges two already sorted lists, to produce a sorted list result. The merge algorithm is really a pattern of computation that can be adapted and reused for various other scenarios, such as finding words that are in a book, but not in a vocabulary.

probe Each time we take a look when searching for an item is called a probe. In our chapter on *Iteration* we also played a guessing game where the computer tried to guess the user's secret number. Each of those tries would also be called a probe.

test-driven development (TDD) A software development practice which arrives at a desired feature through a series of small, iterative steps motivated by automated tests which are *written first* that express increasing refinements of the desired feature. (see the Wikipedia article on Test-driven development for more information.)

14.11 Exercises

1. The section in this chapter called Alice in Wonderland, again! started with the observation that the merge algorithm uses a pattern that can be reused in other situations. Adapt the merge algorithm to write each of these functions, as was suggested there:

   (a) Return only those items that are present in both lists.

   (b) Return only those items that are present in the first list, but not in the second.

   (c) Return only those items that are present in the second list, but not in the first.

   (d) Return items that are present in either the first or the second list.

   (e) Return items from the first list that are not eliminated by a matching element in the second list. In this case, an item in the second list "knocks out" just one matching item in the first list. This operation is sometimes called *bagdiff*. For example `bagdiff([5,7,11,11,11,12,13], [7,8,11])` would return `[5,11,11,12,13]`

2. Modify the queens program to solve some boards of size 4, 12, and 16. What is the maximum size puzzle you can usually solve in under a minute?

3. Adapt the queens program so that we keep a list of solutions that have already printed, so that we don't print the same solution more than once.

4. Chess boards are symmetric: if we have a solution to the queens problem, its mirror solution — either flipping the board on the X or in the Y axis, is also a solution. And giving the board a 90 degree, 180 degree, or 270 degree rotation is also a solution. In some sense, solutions that are just mirror images or rotations of other solutions — in the same family — are less interesting than the unique "core cases". Of the 92 solutions for

the 8 queens problem, there are only 12 unique families if you take rotations and mirror images into account. Wikipedia has some fascinating stuff about this.

(a) Write a function to mirror a solution in the Y axis,

(b) Write a function to mirror a solution in the X axis,

(c) Write a function to rotate a solution by 90 degrees anti-clockwise, and use this to provide 180 and 270 degree rotations too.

(d) Write a function which is given a solution, and it generates the family of symmetries for that solution. For example, the symmetries of `[0,4,7,5,2,6,1,3]` are

```
[[0,4,7,5,2,6,1,3],[7,1,3,0,6,4,2,5],
 [4,6,1,5,2,0,3,7],[2,5,3,1,7,4,6,0],
 [3,1,6,2,5,7,4,0],[0,6,4,7,1,3,5,2],
 [7,3,0,2,5,1,6,4],[5,2,4,6,0,3,1,7]]
```

(e) Now adapt the queens program so it won't list solutions that are in the same family. It only prints solutions from unique families.

5. Every week a computer scientist buys four lotto tickets. She always chooses the same prime numbers, with the hope that if she ever hits the jackpot, she will be able to go onto TV and Facebook and tell everyone her secret. This will suddenly create widespread public interest in prime numbers, and will be the trigger event that ushers in a new age of enlightenment. She represents her weekly tickets in Python as a list of lists:

```
my_tickets = [ [ 7, 17, 37, 19, 23, 43],
               [ 7,  2, 13, 41, 31, 43],
               [ 2,  5,  7, 11, 13, 17],
               [13, 17, 37, 19, 23, 43] ]
```

Complete these exercises.

(a) Each lotto draw takes six random balls, numbered from 1 to 49. Write a function to return a lotto draw.

(b) Write a function that compares a single ticket and a draw, and returns the number of correct picks on that ticket:

```
test(lotto_match([42,4,7,11,1,13], [2,5,7,11,13,17]) == 3)
```

(c) Write a function that takes a list of tickets and a draw, and returns a list telling how many picks were correct on each ticket:

```
test(lotto_matches([42,4,7,11,1,13], my_tickets) == [1,2,3,1])
```

(d) Write a function that takes a list of integers, and returns the number of primes in the list:

```
test(primes_in([42, 4, 7, 11, 1, 13]) == 3)
```

(e) Write a function to discover whether the computer scientist has missed any prime numbers in her selection of the four tickets. Return a list of all primes that she has missed:

```
test(prime_misses(my_tickets) == [3, 29, 47])
```

(f) Write a function that repeatedly makes a new draw, and compares the draw to the four tickets.

   i. Count how many draws are needed until one of the computer scientist's tickets has at least 3 correct picks. Try the experiment twenty times, and average out the number of draws needed.

   ii. How many draws are needed, on average, before she gets at least 4 picks correct?

   iii. How many draws are needed, on average, before she gets at least 5 correct? (Hint: this might take a while. It would be nice if you could print some dots, like a progress bar, to show when each of the 20 experiments has completed.)

   Notice that we have difficulty constructing test cases here, because our random numbers are not deterministic. Automated testing only really works if you already know what the answer should be!

6. Read *Alice in Wonderland*. You can read the plain text version we have with this textbook, or if you have e-book reader software on your PC, or a Kindle, iPhone, Android, etc. you'll be able to find a suitable version for your device at http://www.gutenberg.org/. They also have html and pdf versions, with pictures, and thousands of other classic books!

ch.14

October 11, 2020

0.1 Ch. 14: Algorithms

```
[6]: import math
     import turtle
     import random
     from unit_tester import test
     import time
```

#Linear Search Algorithm

```
[7]: def search_linear(xs, target):
         """ Find and return the index of target in sequence xs """
         for (i, v) in enumerate(xs):
             if v == target:
                 return i
         return -1

     friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
     test(search_linear(friends, "Zoe") == 1)
     test(search_linear(friends, "Joe") == 0)
     test(search_linear(friends, "Paris") == 6)
     test(search_linear(friends, "Bill") == -1)
```

```
Test at line 9 ok.
Test at line 10 ok.
Test at line 11 ok.
Test at line 12 ok.
```

#Create a function that finds unknown words

```
[ ]: def find_unknown_words(vocab, wds):
         """ Return a list of words in wds that do not occur in vocab """
         result = []
         for w in wds:
             if (search_linear(vocab, w) < 0):                   #liner search
             #if (search_binary(vocab, w) < 0):                    #binary search
                 result.append(w)
         return result
```

```
vocab = ["apple", "boy", "dog", "down", "fell", "girl", "grass", "the", "tree"]
book_words = "the apple fell from the tree to the grass".split()
```

```
[8]: test(find_unknown_words(vocab, book_words) == ["from", "to"])
     test(find_unknown_words([], book_words) == book_words)
     test(find_unknown_words(vocab, ["the", "boy", "fell"]) == [])
```

```
Test at line 1 FAILED.
Test at line 2 ok.
Test at line 3 ok.
```

#Load a list of Vocabulary words

```
[10]: def load_words_from_file(filename):
          """ Read words from filename, return list of words. """
          f = open(filename, "r")
          file_content = f.read()
          f.close()
          wds = file_content.split()
          return wds

      bigger_vocab = load_words_from_file("vocab.txt")
      print("There are {0} words in the vocab, starting with\n {1} ".
       ↪format(len(bigger_vocab), bigger_vocab[:6]))
```

```
There are 19455 words in the vocab, starting with
 ['a', 'aback', 'abacus', 'abandon', 'abandoned', 'abandonment']
```

#Translate Function (remove punctuation)

```
[11]: def text_to_words(the_text):
          """ return a list of words with all punctuation removed,
          and all in lowercase.
          """

          my_substitutions = the_text.maketrans(
          # If you find any of these
          "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!\"#$%&()*+,-./:;<=>?@[]^_`{|}~'\\",
          # Replace them by these
          "abcdefghijklmnopqrstuvwxyz                                          ")

          # Translate the text now.
          cleaned_text = the_text.translate(my_substitutions)
          wds = cleaned_text.split()
          return wds
```

```
[12]: test(text_to_words("My name is Earl!") == ["my", "name", "is", "earl"])
```

```
test(text_to_words('"Well, I never!", said Alice.') == ["well", "i", "never",␣
↪"said", "alice"])
```

```
Test at line 1 ok.
Test at line 2 ok.
```

#Load Alice in Wonderland Book

[13]:
```
def get_words_in_book(filename):
    """ Read a book from filename, and return a list of its words. """
    f = open(filename, "r")
    content = f.read()
    f.close()
    wds = text_to_words(content)
    return wds

book_words = get_words_in_book("AliceInWonderland.txt")
print("There are {0} words in the book, the first 100 are\n{1}" .
↪format(len(book_words), book_words[:100]))
```

```
There are 27803 words in the book, the first 100 are
["alice's", 'adventures', 'in', 'wonderland', 'lewis', 'carroll', 'chapter',
'i', 'down', 'the', 'rabbit', 'hole', 'alice', 'was', 'beginning', 'to', 'get',
'very', 'tired', 'of', 'sitting', 'by', 'her', 'sister', 'on', 'the', 'bank',
'and', 'of', 'having', 'nothing', 'to', 'do', 'once', 'or', 'twice', 'she',
'had', 'peeped', 'into', 'the', 'book', 'her', 'sister', 'was', 'reading',
'but', 'it', 'had', 'no', 'pictures', 'or', 'conversations', 'in', 'it', "'and",
'what', 'is', 'the', 'use', 'of', 'a', 'book', "'", 'thought', 'alice',
"'without", 'pictures', 'or', 'conversation', "'", 'so', 'she', 'was',
'considering', 'in', 'her', 'own', 'mind', 'as', 'well', 'as', 'she', 'could',
'for', 'the', 'hot', 'day', 'made', 'her', 'feel', 'very', 'sleepy', 'and',
'stupid', 'whether', 'the', 'pleasure', 'of', 'making']
```

#Compare List of Vocabulary Words to Alice in Wonderland Book

[14]:
```
missing_words = find_unknown_words(bigger_vocab, book_words)
#print(missing_words)
```

#Time the Search

[15]:
```
import time

t0 = time.perf_counter()
missing_words = find_unknown_words(bigger_vocab, book_words)
t1 = time.perf_counter()

print("There are {0} unknown words.".format(len(missing_words)))
print("That took {0:.4f} seconds.".format(t1-t0))
```

```
There are 5310 unknown words.
That took 72.2800 seconds.
```

#Binary Search

[16]:
```
def search_binary(xs, target):
    """ Find and return the index of key in sequence xs """
    lb = 0
    ub = len(xs)
    while True:
        if lb == ub: # If region of interest (ROI) becomes empty
            return -1

        # Next probe should be in the middle of the ROI
        mid_index = (lb + ub) // 2

        # Fetch the item at that position
        item_at_mid = xs[mid_index]

        #print("ROI[{0}:{1}](size={2}), probed='{3}', target='{4}'" .format(lb,␣
 ↪ub, ub-lb, item_at_mid, target))

        # How does the probed item compare to the target?
        if item_at_mid == target:
            return mid_index # Found it!
        if item_at_mid < target:
            lb = mid_index + 1 # Use upper half of ROI next time
        else:
            ub = mid_index # Use lower half of ROI next time
```

#Tests for Binary Search

[17]:
```
xs = [2,3,5,7,11,13,17,23,29,31,37,43,47,53]
test(search_binary(xs, 3) == 1)
test(search_binary(xs, 20) == -1)
test(search_binary(xs, 99) == -1)
test(search_binary(xs, 1) == -1)
for (i, v) in enumerate(xs):
    test(search_binary(xs, v) == i)
```

```
Test at line 4 ok.
Test at line 5 ok.
Test at line 6 ok.
Test at line 7 ok.
Test at line 9 ok.
Test at line 9 ok.
Test at line 9 ok.
Test at line 9 ok.
Test at line 9 ok.
Test at line 9 ok.
Test at line 9 ok.
Test at line 9 ok.
Test at line 9 ok.
Test at line 9 ok.
Test at line 9 ok.
Test at line 9 ok.
Test at line 9 ok.
Test at line 9 ok.
```

#Test Time again compared to algorithm "search_linear"

[18]:
```python
import time

t0 = time.perf_counter()
missing_words = find_unknown_words(bigger_vocab, book_words)
t1 = time.perf_counter()

print("There are {0} unknown words.".format(len(missing_words)))
print("That took {0:.4f} seconds.".format(t1-t0))
```

```
There are 5310 unknown words.
That took 73.3399 seconds.
```

#Remove Duplicates

[ ]:
```python
def remove_adjacent_dups(xs):
    """ Return a new list in which all adjacent
    duplicates from xs have been removed.
    """
    result = []
    most_recent_elem = None
    for e in xs:
        if e != most_recent_elem:
            result.append(e)
            most_recent_elem = e

    return result
```

#Merge Sorted Lists

[ ]:
```python
def merge(xs, ys):
    """ merge sorted lists xs and ys. Return a sorted result """
    result = []
    xi = 0
    yi = 0
    while True:
        if xi >= len(xs): # If xs list is finished,
            result.extend(ys[yi:]) # Add remaining items from ys
            return result # And we're done.

        if yi >= len(ys): # Same again, but swap roles
            result.extend(xs[xi:])
            return result

        # Both lists still have items, copy smaller item to result.
        if xs[xi] <= ys[yi]:
            result.append(xs[xi])
            xi += 1
        else:
            result.append(ys[yi])
            yi += 1
```

[20]:
```
xs = [1,3,5,7,9,11,13,15,17,19]
ys = [4,8,12,16,20,24]
zs = xs+ys
zs.sort()
test(merge(xs, []) == xs)
test(merge([], ys) == ys)
test(merge([], []) == [])
test(merge(xs, ys) == zs)

test(merge([1,2,3], [3,4,5]) == [1,2,3,3,4,5])
test(merge(["a", "big", "cat"], ["big", "bite", "dog"]) == ["a", "big", "big",␣
 ↪"bite", "cat", "dog"])
```

```
Test at line 5 ok.
Test at line 6 ok.
Test at line 7 ok.
Test at line 8 ok.
Test at line 10 ok.
Test at line 11 ok.
```

#Merge sorted Alice in Wonderland Example

[ ]:
```
def find_unknowns_merge_pattern(vocab, wds):
    """ Both the vocab and wds must be sorted. Return a new
    list of words from wds that do not occur in vocab.
    """

    result = []
    xi = 0
    yi = 0
    while True:
        if xi >= len(vocab):
            result.extend(wds[yi:])
            return result

        if yi >= len(wds):
            return result

        if vocab[xi] == wds[yi]: # Good, word exists in vocab
            yi += 1

        elif vocab[xi] < wds[yi]: # Move past this vocab word,
            xi += 1

        else: # Got word that is not in vocab
            result.append(wds[yi])
            yi += 1
```

[ ]:
```
#Test
```

[21]:
```
all_words = get_words_in_book("AliceInWonderland.txt")
#t0 = time.perf_counter()
all_words.sort()

book_words = remove_adjacent_dups(all_words)

missing_words = find_unknowns_merge_pattern(bigger_vocab, book_words)

t1 = time.perf_counter()

print("There are {0} unknown words.".format(len(missing_words)))
print("That took {0:.4f} seconds.".format(t1-t0))

#Debuger get nicht weiter als Zeile 277!
```

```
There are 1133 unknown words.
That took 246.5545 seconds.
```

0.2 Ch. 14: Merge Algorithm for merging Lists

#Ex. 1

#a)

[24]:
```
def return_both_present(xs, ys):
    """ merge sorted lists xs and ys. Return a sorted result """
    result = []
    xi = 0
    yi = 0

    while True:
        if xi >= len(xs):
            return result

        if yi >= len(ys):
            return result

        if xs[xi] < ys[yi]:
            xi += 1
        elif xs[xi] > ys[yi]:
            yi += 1
        else:
            result.append(xs[xi])
            xi += 1
            yi += 1
```

[25]:
```
print(return_both_present([1, 1, 3, 5, 7, 8, 9, 10], [1, 2, 3, 4, 5, 6, 7, 10]))
```

```
[1, 3, 5, 7, 10]
```

#b)

[26]:
```
def return_first_list_present_only(xs, ys):
    """ merge sorted lists xs and ys. Return a sorted result """
    result = []
    xi = 0
    yi = 0

    while True:
        if xi >= len(xs):
            return result

        if yi >= len(ys):
            result.append(xs[yi:])
            return result

        if xs[xi] < ys[yi]:
            result.append(xs[xi])
            xi += 1
        elif xs[xi] == ys[yi]:
            xi += 1
        else:
            yi += 1
```

[27]:
```
print(return_first_list_present_only([0, 1, 1, 2, 3, 4, 5, 7], [1, 3, 5, 6, 7,␣
 ↪10]))
```

```
[0, 2, 4]
```

#c)

[28]:
```python
def return_second_list_present_only(xs, ys):
    """ merge sorted lists xs and ys. Return a sorted result """
    result = []
    xi = 0
    yi = 0

    while True:
        if xi >= len(xs):
            result.extend(ys[yi:])
            return result

        if yi >= len(ys):

            return result

        if xs[xi] < ys[yi]:
            xi += 1
        elif xs[xi] == ys[yi]:
            yi += 1
        else:
            result.append(ys[yi])
            yi += 1
```

[29]:
```python
print(return_second_list_present_only([0, 1, 1, 2, 3, 4, 5, 7], [-1, 1, 3, 5, 6,␣
 ↪7, 8, 10]))
```

```
[-1, 6, 8, 10]
```

#d)

[30]:
```python
def return_unique_items_in_both_lists(xs, ys):
    """ merge sorted lists xs and ys. Return a sorted result """
    result = []
    xi = 0
    yi = 0

    while True:
        if xi >= len(xs):
            result.extend(ys[yi:])
            return result

        if yi >= len(ys):
            result.extend(xs[xi:])
            return result
        if xs[xi] < ys[yi]:
            result.append(xs[xi])
            xi += 1
        elif xs[xi] > ys[yi]:
            result.append(ys[yi])
            yi += 1
        else:
            xi += 1
            yi += 1
```

[31]:
```
print(return_unique_items_in_both_lists([0, 1, 2.5, 3, 4, 4.5, 5, 7], [-1, 1, 2,␣
 ↪3, 5, 6, 7, 8, 10]))
```

```
[-1, 0, 2, 2.5, 4, 4.5, 6, 8, 10]
```

#e)

[32]:
```
def bagdiff(xs, ys):
    """ merge sorted lists xs and ys. Return a sorted result """
    result = []
    xi = 0
    yi = 0

    while True:
        if xi >= len(xs):
            result.extend(ys[yi:])
            return result

        if yi >= len(ys):
            result.extend(xs[xi:])
            return result

        if xs[xi] < ys[yi]:
            result.append(xs[xi])
            xi += 1
        elif xs[xi] > ys[yi]:
            yi += 1
        else:
            xi += 1
            yi += 1
```

[33]:
```
print(bagdiff([5, 7, 11, 11, 11, 12, 13], [7, 8, 11]))
from unit_tester import test

test(bagdiff([5,7,11,11,11,12,13], [7,8,11]) == [5,11,11,12,13])
```

```
[5, 11, 11, 12, 13]
Test at line 4 ok.
```

#Ex. 2, 3, 4 #skipped

#Ex. 5: Lottery with Prime Numbers

[34]:
```
import random
from unit_tester import test

my_tickets = [[7, 17, 37, 19, 23, 43], [7, 2, 13, 41, 31, 43], [2, 5, 7, 11, 13,␣
 ↪17],
              [13, 17, 37, 19, 23, 43]]


def lotto_draw():
    lotto_generator = random.Random()
    result = []
    for i in range(6):
        result.append(lotto_generator.uniform(1, 50))
    return result


def lotto_match(lotto1, lotto2):
    count = 0
    for item in lotto1:
        if item in lotto2:
            count += 1
    return count


def lotto_matches(lotto, mytick):
    result = []
    for i in range(4):
        result.append(lotto_match(lotto, my_tickets[i]))
    return result


def PriNumGenerator(upperlimit):
    Plist = [2]
    for num in range(2, upperlimit + 1):
        isprime = True
        for i in Plist:
            if num % i == 0:
                isprime = False
                break
        if isprime == True:
            Plist.append(num)
    return Plist
```

```
def primes_in(l):
    prime_list = PriNumGenerator(50)
    count = 0
    for item in l:
        if item in prime_list:
            count += 1
    return count


def prime_misses(l):
    prime_list = PriNumGenerator(50)
    result = []
    new = []
    for i in range(len(l)):
        new += l[i]
    for item in prime_list:
        if item in new:
            continue
        else:
            result.append(item)
    return result
```

[23]:
```
test(lotto_match([42, 4, 7, 11, 1, 13], [2, 5, 7, 11, 13, 17]) == 3)
test(lotto_matches([42, 4, 7, 11, 1, 13], my_tickets) == [1, 2, 3, 1])
test(primes_in([42, 4, 7, 11, 1, 13]) == 3)
test(prime_misses(my_tickets) == [3, 29, 47])
```

```
Test at line 1 ok.
Test at line 2 ok.
Test at line 3 ok.
Test at line 4 ok.
```

[ ]:
```

```