BIOS6642

Algorithm Quality

Fuyong Xing

Department of Biostatistics and Informatics

Colorado School of Public Health

University of Colorado Anschutz Medical Campus

Outline

- **Good algorithms versus bad algorithms**
- Sorting
- Search
- Memoization

Good algorithms versus bad algorithms

- Program correctness is the primary goal of software construction, but correctness is not the only goal.
- Two different programs may produce the exact same results in all cases, but one may objectively be considered better than the other.
  - For example, one is faster than the other.

Good algorithms versus bad algorithms

- Suppose we want to determine if the elements in an integer list appear in ascending order.

Codes

```
def is_ascending(lst):
    for i in range(len(lst) - 1):
        for j in range(i + 1, len(lst)):
            if lst[i] > lst[j]:
                return False
    return True

lst1 = [2,1,9,0,8,6]
print('List', lst1, ':', is_ascending(lst1))
lst2 = [0,1,2,6,8,9]
print('List', lst2, ':', is_ascending(lst2))
```

Output

```
List [2, 1, 9, 0, 8, 6] : False
List [0, 1, 2, 6, 8, 9] : True
```

Good algorithms versus bad algorithms

- The *is_ascending* function first compares the first (*0*-th index) element to all the elements that follow.
- If the first element is smaller than all the others, it continues by comparing the second element to all of the elements that follow the second element.
- The third time through the outer loop the function compares the third element to all the elements that follow the third element.
- If at any time the the inner loop finds an element smaller than the element controlled by the outer loop, the function immediately returns *False*.
- If the elements in the list are in ascending order, the nested loop must continue until it finally compares the next-to-the-last element to the last element. At that point the function returns *True*.

Good algorithms versus bad algorithms

- Consider another algorithm that also determines if the elements in an integer list appear in ascending order.

Codes

```
def is_ascending2(lst):
    for i in range(len(lst) - 1):
        if lst[i] > lst[i + 1]:
            return False
    return True

lst1 = [2,1,9,0,8,6]
print('List', lst1, ':', is_ascending2(lst1))
lst2 = [0,1,2,6,8,9]
print('List', lst2, ':', is_ascending2(lst2))
```

Output

```
List [2, 1, 9, 0, 8, 6] : False
List [0, 1, 2, 6, 8, 9] : True
```

Good algorithms versus bad algorithms

- The is_ascending2 function compares the first element to the second element, the second element to the third, the third to the fourth, etc., until it finally compares the last second element to the last element.
- If the function detects any element out of order, it returns *False* immediately.
- If it makes it all the way to the end of the list, the function returns *True*.

Good algorithms versus bad algorithms

- Which one is better?

```
def is_ascending(lst):
    for i in range(len(lst) - 1):
        for j in range(i + 1, len(lst)):
            if lst[i] > lst[j]:
                return False
    return True
```

```
def is_ascending2(lst):
    for i in range(len(lst) - 1):
        if lst[i] > lst[i + 1]:
            return False
    return True
```

Good algorithms versus bad algorithms

- We can analyze the computational complexity of the two algorithms.
- For a list containing *n* elements
  - The *is_ascending* function needs to perform the comparison in the *if* statement up to ***n(n-1)/2*** times.
  - The *is_ascending2* function needs to perform the comparison in the *if* statement up to ***n-1*** times.
- What if *n=5000*?

Good algorithms versus bad algorithms

- Analysis of algorithms: the process of finding the computational complexity of algorithms, such as the amount of time, storage, or other resources needed to execute them.
- Analysis of algorithms is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm that solves a given computational problem.
- The practical goal of algorithm analysis is to predict the performance of different algorithms in order to guide design decisions.

https://en.wikipedia.org/wiki/Analysis_of_algorithms

Good algorithms versus bad algorithms

- An experiment to compare the *is_ascending* and *is_ascending2* functions.

Codes

```
from time import clock
from math import sqrt

def is_ascending(lst):
    """ Returns True if lst contains elements
        in nondecreasing order as determined by
        the < operator.  Returns False if the
        elements of lst are not in order.  Throws
        an exception if lst is not a list or its
        elements are not compatible with the <
        operator. """
    for i in range(len(lst) - 1):
        for j in range(i + 1, len(lst)):
            if lst[i] > lst[j]:
                return False
    return True
```

Codes

```
def is_ascending2(lst):
    """ Returns True if lst contains elements
        in nondecreasing order as determined by
        the < operator.  Returns False if the
        elements of lst are not in order.  Throws
        an exception if lst is not a list or its
        elements are not compatible with the <
        operator. """
    for i in range(len(lst) - 1):
        if lst[i] > lst[i + 1]:
            return False
    return True


def compute_time(size, data1, data2):
    """ Compares the performance of the is_ascending and
        is_ascending2 functions on a list of a given size.
        Prints the results of the executions and appends the
        results to the data1 and data2 lists for further processing. """

    print("List size:", size)
```

Codes

```
    my_list = list(range(size))   #  Make list [0, 1, 2, 3, ..., size - 1]
    start = clock()               # Start the clock
    ans = is_ascending(my_list)   # Compute answer
    elapsed1 = clock() - start    # Stop the clock
    print("  is_ascending:  {} Elapsed: {:12.7f}".format(ans, elapsed1))
    start = clock()               # Start the clock
    ans = is_ascending2(my_list)  # Compute answer
    elapsed2 = clock() - start    # Stop the clockt
    print("  is_ascending2: {} Elapsed: {:12.7f}".format(ans, elapsed2))
    print("  Speedup: {:6.1f}".format(elapsed1/elapsed2)) # Compute speedup
    print()
    data1.append((size, elapsed1))
    data2.append((size, elapsed2))


def main():
    """ Compares the performance of the is_ascending and
        is_ascending2 functions on lists of various sizes. """
    data1, data2 = [], []

    # Compute results for sizes in the range 0...40,000
```

Codes

```
    max_size = 40000
    # Sizes used are 0**2 = 0, 20**2 = 400, 40**2 = 1600, 60**2 = 3600,
    # etc. up to 200**2 = 40,000
    for size in (x**2 for x in range(0, round(sqrt(max_size)) + 1, 20)):
        compute_time(size, data1, data2)


if __name__ == '__main__':
    main()
```

Output

```
List size: 0
  is_ascending:  True Elapsed:    0.0000026
  is_ascending2: True Elapsed:    0.0000154
  Speedup:    0.2

List size: 400
  is_ascending:  True Elapsed:    0.0105160
  is_ascending2: True Elapsed:    0.0000727
  Speedup:  144.6

List size: 1600
  is_ascending:  True Elapsed:    0.1632568
  is_ascending2: True Elapsed:    0.0002386
```

Output

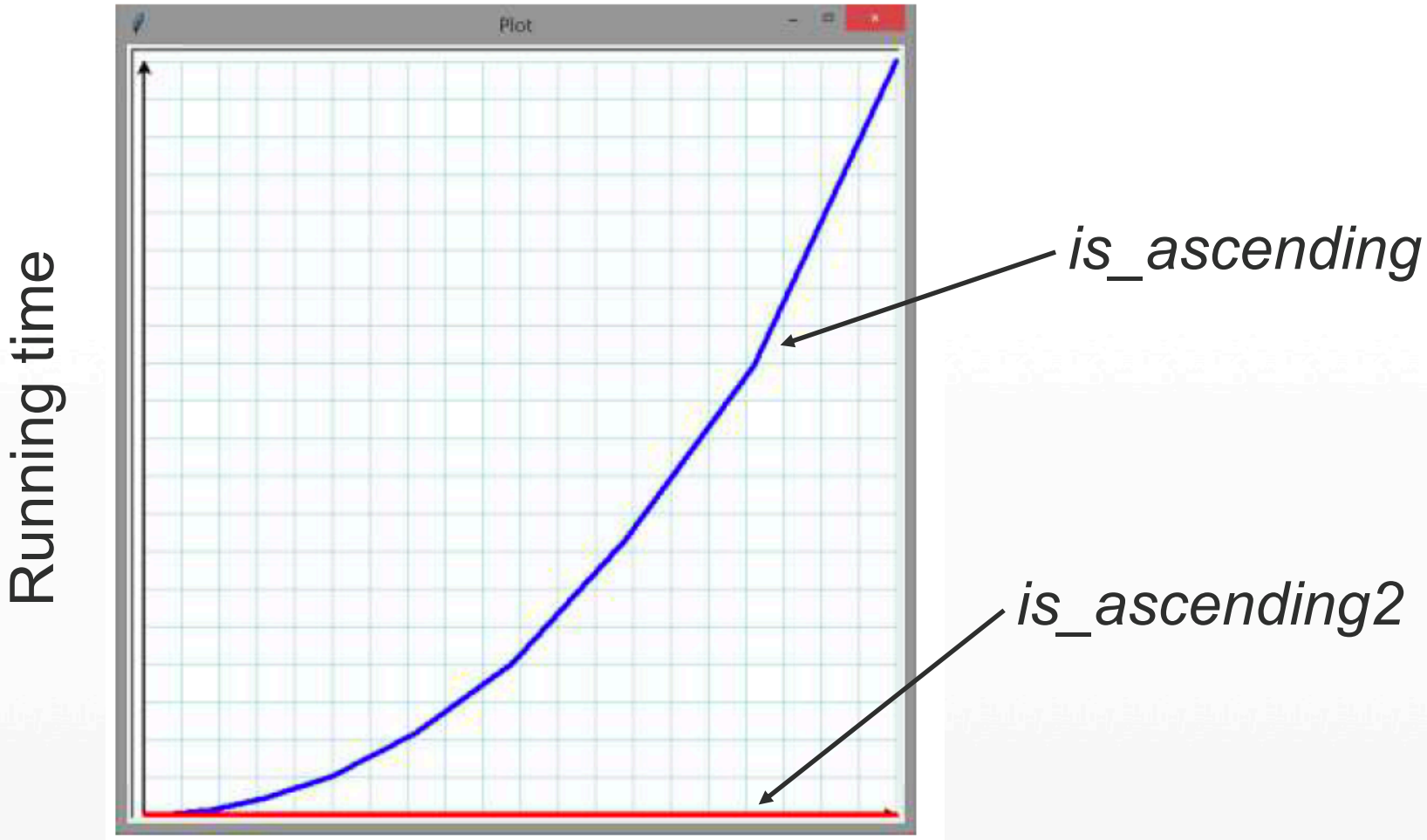
```
  Speedup:  684.1

List size: 3600
  is_ascending:  True Elapsed:    0.8197846
  is_ascending2: True Elapsed:    0.0005183
  Speedup: 1581.6

List size: 6400
  is_ascending:  True Elapsed:    2.5957844
  is_ascending2: True Elapsed:    0.0009212
  Speedup: 2817.9

List size: 10000
  is_ascending:  True Elapsed:    6.3051484
  is_ascending2: True Elapsed:    0.0014296
  Speedup: 4410.3

List size: 14400
  is_ascending:  True Elapsed:   13.0304075
  is_ascending2: True Elapsed:    0.0020309
  Speedup: 6416.0
```

Output

```
List size: 19600
  is_ascending:  True Elapsed:   24.1843773
  is_ascending2: True Elapsed:    0.0027695
  Speedup: 8732.4

List size: 25600
  is_ascending:  True Elapsed:   43.8347709
  is_ascending2: True Elapsed:    0.0060735
  Speedup: 7217.3

List size: 32400
  is_ascending:  True Elapsed:   71.5207196
  is_ascending2: True Elapsed:    0.0076400
  Speedup: 9361.3

List size: 40000
  is_ascending:  True Elapsed:  120.1243236
  is_ascending2: True Elapsed:    0.0062690
  Speedup: 19161.7
```

Good algorithms versus bad algorithms

- Running time of the *is_ascending* (blue) and *is_ascending2* (red) functions.



Outline

- Good algorithms versus bad algorithms
- **Sorting**
- Search
- Memoization

Sorting

- Sort a list: arranging the elements within a list into a particular order.
- Python lists have a sort method for sorting: *list.sort*.
- Python also provides a built-in sorting function, *sorted*, that can be used to sort a list. It returns a new sorted list.

Codes

```
for elem in sorted([34, 2, 22, 70, 16, 8]):
    print(elem, end=" ")
print()
```

Output

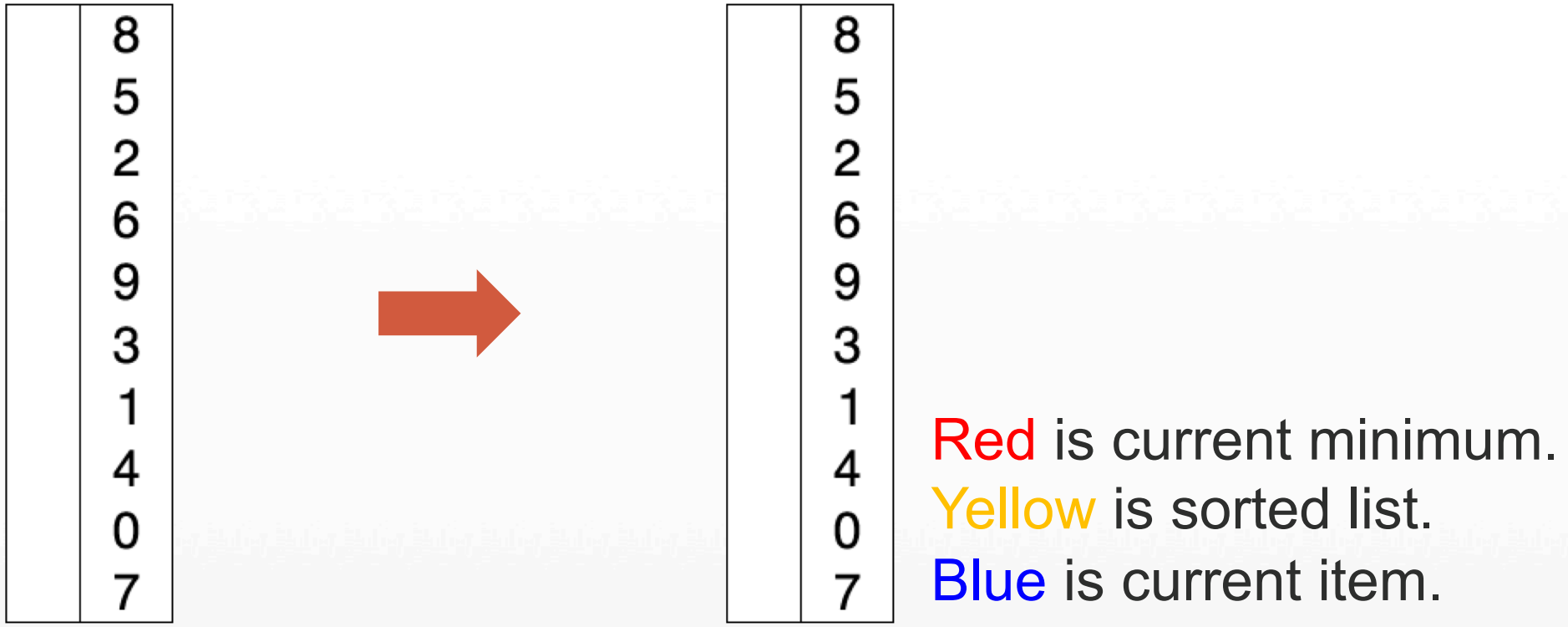
```
2 8 16 22 34 70
```

Sorting

- Selection sort algorithm
  1. Set *n* = *length of list A*.
  2. Set *i* = *0*.
  3. Examine all the elements *A[j]*, where $i < j < n$. If any of these elements is less than *A[i]*, then exchange *A[i]* with the smallest of these elements. (This ensures that all elements after position *i* are greater than or equal to *A[i]*.)
  4. If *i* is less than $n-1$, increase *i* by *1* and go to Step 3.
  5. Done; list *A* is sorted.

Sorting

- Selection sort algorithm
  - Suppose A=[8,5,2,6,9,3,1,4,0,7]



See the gif figure via this link:
https://en.wikipedia.org/wiki/Selection_sort

Sorting

- Selection sort

Codes

```
from random import randint

def random_list():
    """
    Produce a list of pseudorandom integers.
    The list's length is chosen pseudorandomly in the
    range 3-20.
    The integers in the list range from -50 to 50.
    """
    result = []
    count = randint(3, 20)
    for i in range(count):
        result.append(randint(-50, 50))
    return result
```

Codes

```
def selection_sort(lst):
    """
    Arranges the elements of list lst in ascending order.
    Physically rearranges the elements of lst.
    """
    n = len(lst)
    for i in range(n - 1):
        # Note: i, small, and j represent positions within lst
        # lst[i], lst[small], and lst[j] represent the elements at
        # those positions.
        # small is the position of the smallest value we've seen
        # so far; we use it to find the smallest value less
        # than lst[i]
        small = i
        # See if a smaller value can be found later in the list
        # Consider all the elements at position j, where i < j < n
        for j in range(i + 1, n):
            if lst[j] < lst[small]:
                small = j       # Found a smaller value
        # Swap lst[i] and lst[small], if a smaller value was found
        if i != small:
            lst[i], lst[small] = lst[small], lst[i]
```

Codes

```
def main():
    """
    Tests the selection_sort function
    """
    for n in range(10):
        col = random_list()
        print(col)
        selection_sort(col)
        print(col)
        print('================================')

main()
```

Output

```
[-23, 47, -3, 4, 5, -46, 26, -27]
[-46, -27, -23, -3, 4, 5, 26, 47]
================================
[32, -10, -4, 41, 10, -1, -31, 3, 28, -31, -33, 46, -45, -6, 37]
[-45, -33, -31, -31, -10, -6, -4, -1, 3, 10, 28, 32, 37, 41, 46]
================================
[11, -19, 20, 43, -19, 20, -18, -17]
[-19, -19, -18, -17, 11, 20, 20, 43]
================================
```

Output

```
[9, -22, -41, 35, 10, 48, 9, 14, -20]
[-41, -22, -20, 9, 9, 10, 14, 35, 48]
==============================
[-38, -3, -7, 41, -8, -11, -23, 9, -47, 38]
[-47, -38, -23, -11, -8, -7, -3, 9, 38, 41]
==============================
[-47, 1, -37, 16, -40, -14, 2, 38, 43, 19, 45]
[-47, -40, -37, -14, 1, 2, 16, 19, 38, 43, 45]
==============================
[8, 39, 35, -42]
[-42, 8, 35, 39]
==============================
[-8, -22, -13, 47, -28, -46, -21, -42, 27, 14, 47, -21, 2, -47]
[-47, -46, -42, -28, -22, -21, -21, -13, -8, 2, 14, 27, 47, 47]
==============================
[37, -21, -32, -7]
[-32, -21, -7, 37]
==============================
[33, -42, -26, 35, 37, 36, -1, 47, 24, 5, 41, -6, 48, 6, 43]
[-42, -26, -6, -1, 5, 6, 24, 33, 35, 36, 37, 41, 43, 47, 48]
==============================
```

Sorting

- Can we change the behavior of the sorting function so that it arranges the elements in descending order instead of ascending order?
- How can we rewrite the previous *selection_sort* function such that, by passing an additional parameter, it can sort the list in any way we want?
  - We might want to replace the expression in the *if* statement with an ordering function, which is passed as an additional parameter.

Codes

```
def random_list():
    """
    Produce a list of pseudorandom integers.
    The list's length is chosen pseudorandomly in the
    range 3-20.
    The integers in the list range from -50 to 50.
    """
    from random import randrange
    result = []
    count = randrange(3, 20)

    for i in range(count):
        result += [randrange(-50, 50)]
    return result



def less_than(m, n):
    """  Returns true if m is less than n; otherwise, returns false """
    return m < n
```

Codes

```
def greater_than(m, n):
    """  Returns true if m is greater than n; otherwise, returns false """
    return m > n


def selection_sort(lst, cmp):
    """
    Arranges the elements of list lst in ascending order.
    The comparer function cmp is used to order the elements.
    The contents of lst are physically rearranged.
    """
    n = len(lst)
    for i in range(n - 1):
        # Note: i, small, and j represent positions within lst
        # lst[i], lst[small], and lst[j] represent the elements at
        # those positions.
        # small is the position of the smallest value we've seen
        # so far; we use it to find the smallest value less
        # than lst[i]
        small = i
        # See if a smaller value can be found later in the list
        # Consider all the elements at position j, where i < j < n.
```

Codes

```
        for j in range(i + 1, n):
            if cmp(lst[j], lst[small]):
                small = j       # Found a smaller value
        # Swap lst[i] and lst[small], if a smaller value was found
        if i != small:
            lst[i], lst[small] = lst[small], lst[i]

def main():
    """
    Tests the selection_sort function
    """
    original = random_list()        # Make a random list
    working = original[:]           # Make a working copy of the list
    print('Original:  ', working)
    selection_sort(working, less_than)  # Sort ascending
    print('Ascending: ', working)
    working = original[:]           # Make a working copy of the list
    print('Original:  ', working)
    selection_sort(working, greater_than)  # Sort descending
    print('Descending:', working)

main()
```

Output

```
Original:   [-8, 24, -46, -7, -26, -29, -44]
Ascending:  [-46, -44, -29, -26, -8, -7, 24]
Original:   [-8, 24, -46, -7, -26, -29, -44]
Descending: [24, -7, -8, -26, -29, -44, -46]
```

- The comparison function passed to the *selection_sort* function customizes the sort's behavior. The basic structure of the sorting algorithm does not change, but its notion of ordering is adjustable.
- Selection sort is a relatively efficient simple sort, but more advanced sorts are, on average, much faster than selection sort, especially for large data sets.
- More sorting algorithms: https://en.wikipedia.org/wiki/Sorting_algorithm

Outline

- Good algorithms versus bad algorithms
- Sorting
- **Search**
- Memoization

Search

- It is common to search a sequence (e.g., a list) for a particular element.
- Linear search: sequentially checks each element of the list until a match is found or the whole list has been searched.

Codes

```
def locate(lst, seek):
    """
    Returns the index of element seek in list lst,
    if seek is present in lst.
    Returns None if seek is not an element of lst.
    lst is the list in which to search.
    seek is the element to find.
    """
    for i in range(len(lst)):
        if lst[i] == seek:
            return i # Return position immediately
    return None # Element not found
```

Codes

```
def display(lst, value):
    """
    Print the location of value in lst if found.
    """
    position = locate(lst, value)
    if position != None:
        print("Index of ", value, ": ", position, sep='')
    else:
        print(value, " not in list", sep='')
    print('============================')

def main():
    a = [100, 44, 2, 80, 5, 13, 11, 2, 110]
    print(a,'\n') # Print the elements of the list
    display(a, 13)
    display(a, 2)
    display(a, 7)
    display(a, 100)
    display(a, 110)

main()
```

Output

```
[100, 44, 2, 80, 5, 13, 11, 2, 110]

Index of 13: 5
================================
Index of 2: 2
================================
7 not in list
================================
Index of 100: 0
================================
Index of 110: 8
================================
```

- A linear search runs in at worst linear time and makes at most *n* comparisons, where *n* is the length of the list.

Search

- What if the list to be searched is a sorted list?

Search

- What if the list to be searched is a sorted list?
  - A linear search might not be necessary.
- Binary search
  1. If the list is empty, return *None*.
  2. Check the element in the middle of the list. If that element is what you are seeking, return its position. If the middle element is larger than the element you are seeking, perform a binary search on the first half of the list. If the middle element is smaller than the element you are seeking, perform a binary search on the second half of the list.

Codes

```
def binary_search(lst, seek):
    """
    Returns the index of element seek in list lst,
    if seek is present in lst.
    Returns None if seek is not an element of lst.
    lst is the list in which to search.
    seek is the element to find.
    """
    first = 0 # Initialize the first position in list
    last = len(lst) - 1 # Initialize the last position in list
    while first <= last:
        # mid is middle position in the list
        mid = first + (last - first + 1)//2 # Note: Integer division
        if lst[mid] == seek:
            return mid # Found it
        elif lst[mid] > seek:
            last = mid - 1 # continue with 1st half
        else: # v[mid] < seek
            first = mid + 1 # continue with 2nd half
    return None # Not there
```

Codes

```
def display(lst, value):
    """
    Print the location of value in lst if found.
    """
    position = binary_search(lst, value)
    if position != None:
        print("Index of ", value, ": ", position, sep='')
    else:
        print(value, " not in list", sep='')
    print('==============================')

def main():
    a = [10, 14, 20, 28, 29, 33, 34, 45, 48]
    print(a,'\n') # Print the elements of the list
    display(a, 33)
    display(a, 2)
    display(a, 10)
    display(a, 28)
    display(a, 100)
    display(a, 20)

main()
```

Output

```
[10, 14, 20, 28, 29, 33, 34, 45, 48]

Index of 33: 5
================================
2 not in list
================================
Index of 10: 0
================================
Index of 28: 3
================================
100 not in list
================================
Index of 20: 2
================================
```

- A binary search runs in logarithmic time in the worst case, making $log_2 n$ comparisons, where n is the number of elements in the list.

Search

- Binary search versus linear search

Codes

```
def binary_search(lst, seek):
    """
    Returns the index of element seek in list lst,
    if seek is present in lst.
    lst must be in sorted order.
    Returns None if seek is not an element of lst.
    lst is the list in which to search.
    seek is the element to find.
    """
    first = 0              # Initially the first element in list
    last = len(lst) - 1    # Initially the last element in list
    while first <= last:
        # mid is middle of the list
        mid = first + (last - first + 1)//2  # Note: Integer division
        if lst[mid] == seek:
            return mid         # Found it
        elif lst[mid] > seek:
```

Codes

```
            last = mid - 1   # continue with 1st half
        else:  # v[mid] < seek
            first = mid + 1  # continue with 2nd half
    return None    # Not there


def ordered_linear_search(lst, seek):
    """
    Returns the index of element seek in list lst,
    if seek is present in lst.
    lst must be in sorted order.
    Returns None if seek is not an element of lst.
    lst is the list in which to search.
    seek is the element to find.
    """
    i = 0
    n = len(lst)
    while i < n and lst[i] <= seek:
        if lst[i] == seek:
            return i      # Return position immediately
        i += 1
    return None           # Element not found
```

Codes

```
def run_search(lst, seeks, search, trials):
    """
    Searches for all the elements in an ordered list (lst)
    using search function search.  Averages the running time over
    trials runs.  Returns the average time.
    """
    from time import clock
    n = len(lst)
    elapsed = 0
    start = clock()    # Start the clock
    for i in range(trials):
        for elem in seeks:
            i = search(lst, elem)
            if i != lst[i]:
                print("error")
    stop = clock()     # Stop the clock
    elapsed += stop - start
    return elapsed/trials    # Average time for search


def test_searches(lst, seeks, trials):
    """
```

Codes

```
    Measures the running times of ordered linear search and binary
    search on a given list.  Averages the times over n runs.
    """
    # Find each element using ordered linear search
    lin = run_search(lst, seeks, ordered_linear_search, trials)
    # Find each element using binary search
    bin = run_search(lst, seeks, binary_search, trials)
    # Print the results
    print('{0:6} {1:10.5f} {2:10.5f} {3:8.1f}'\
              .format(len(lst), lin, bin, lin/bin))


def make_search_set(n):
    """
    Make a list of elements to seek
    """
    from random import randrange
    result = []
    for i in range(n):
        result += [randrange(n)]
    return result
```

Codes

```
def main():
    """
    Makes a table comparing the running times of ordered linear
    search vs. binary search on lists of various sizes.
    """
    # Number of trials over which to average the results
    trials = 10


    # Print table header
    print('   Size    Linear     Binary    Speedup')
    print('-----------------------------------------')
    # Small lists: 10 to 100, in steps of 10
    for size in range(10, 100, 10):
        test_list = list(range(size))
        seek_list = make_search_set(size)
        test_searches(test_list, seek_list, trials)
    # Medium lists: 100 to 1,000, in steps of 100
    for size in range(100, 1000, 100):
        test_list = list(range(size))
        seek_list = make_search_set(size)
```

Codes

```
        test_searches(test_list, seek_list, trials)
    # Large lists: 1,000 to 5,000, in steps of 500
    for size in range(1000, 5001, 500):
        test_list = list(range(size))
        seek_list = make_search_set(size)
        test_searches(test_list, seek_list, trials)


if __name__ == '__main__':
    main()
```

Output

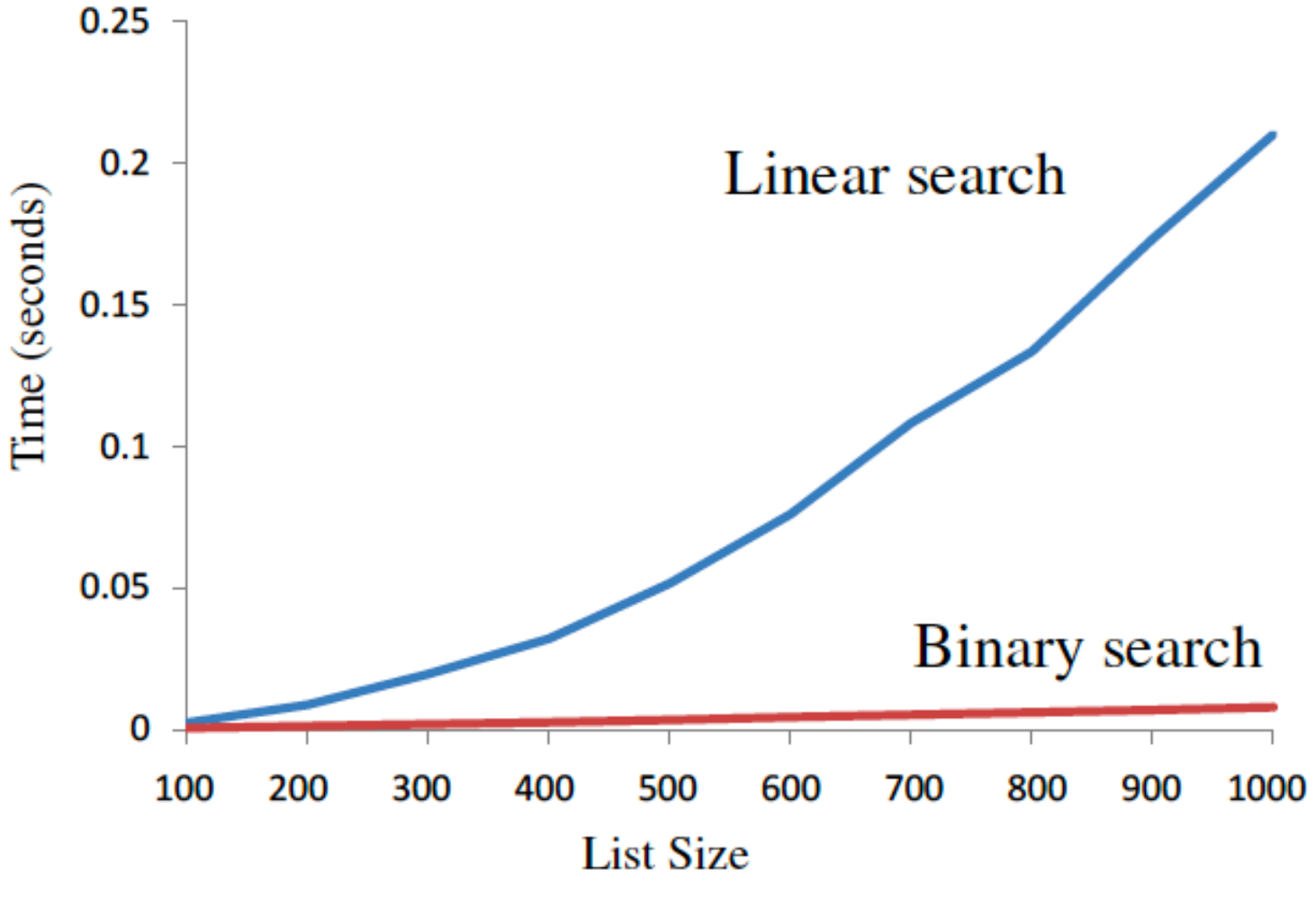
```
   Size     Linear      Binary     Speedup
-----------------------------------------
    10     0.00003     0.00003        1.2
    20     0.00012     0.00006        2.0
    30     0.00024     0.00012        1.9
    40     0.00036     0.00016        2.3
    50     0.00049     0.00021        2.3
    60     0.00082     0.00026        3.1
    70     0.00104     0.00032        3.2
    80     0.00138     0.00039        3.6
```

Output

```
  90     0.00202     0.00043      4.7
 100     0.00245     0.00049      5.0
 200     0.00868     0.00115      7.5
 300     0.01962     0.00185     10.6
 400     0.03215     0.00262     12.3
 500     0.05158     0.00342     15.1
 600     0.07590     0.00437     17.4
 700     0.10805     0.00522     20.7
 800     0.13329     0.00600     22.2
 900     0.17307     0.00687     25.2
1000     0.20985     0.00780     26.9
1500     0.49346     0.01256     39.3
2000     0.85834     0.01739     49.4
2500     1.39785     0.02284     61.2
3000     1.95955     0.02802     69.9
3500     2.71689     0.03321     81.8
4000     3.56608     0.03960     90.1
4500     4.45774     0.04446    100.3
5000     7.73395     0.04983    155.2
```

Search

- Binary search versus linear search



Outline

- Good algorithms versus bad algorithms
- Sorting
- Search
- **Memoization**

Memoization

- Fibonacci sequence: each number is the sum of the two preceding ones, starting from 0 and 1.

$$F_0 = 0 \qquad F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}, \ \textit{for} \ \ n > 1$$

https://en.wikipedia.org/wiki/Fibonacci_number

Memoization

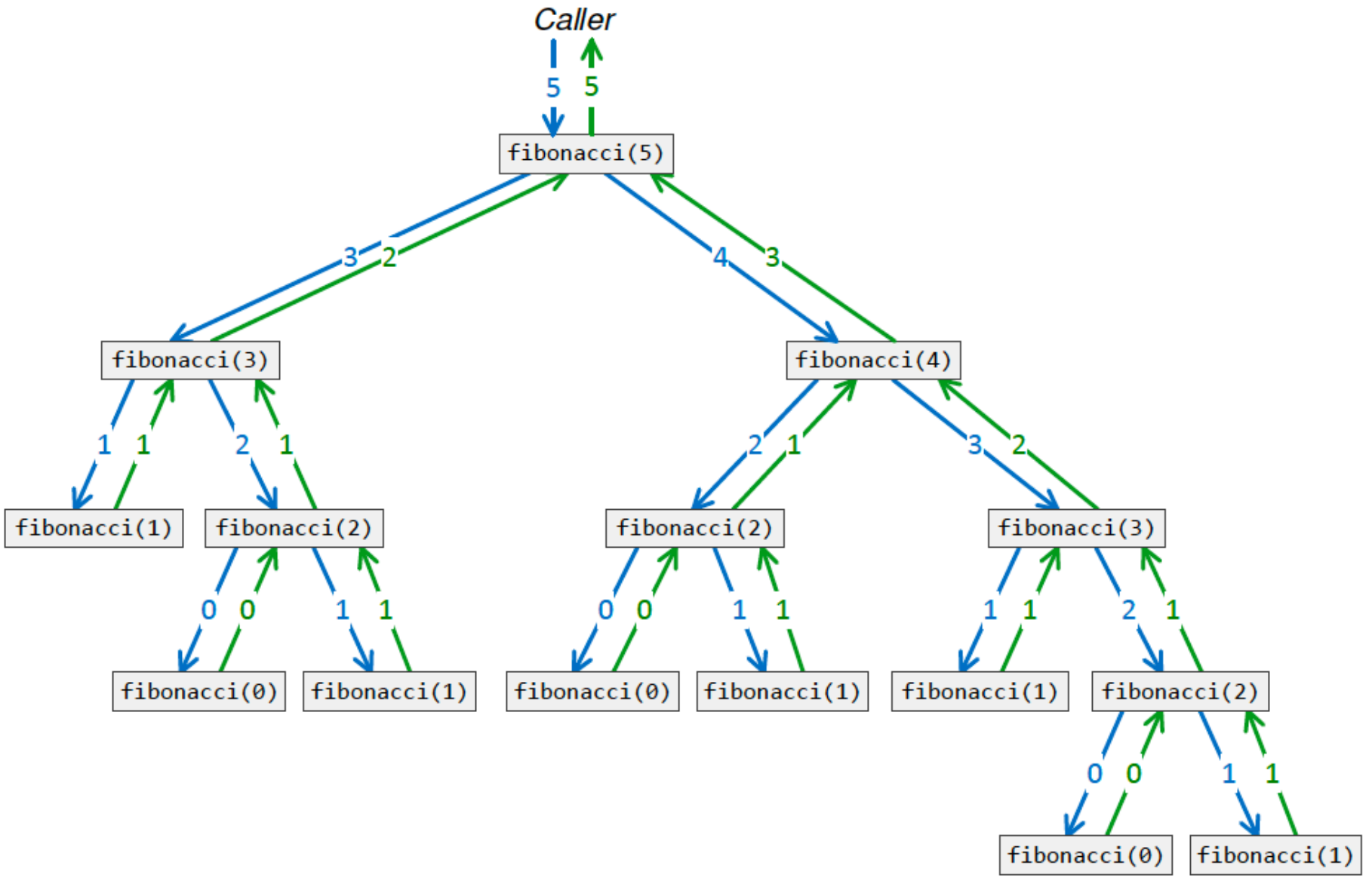
- A recursive Python function to compute the *n*-th Fibonacci number

```
def fibonacci(n):
    """ Returns the nth Fibonacci number. """
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 2) + fibonacci(n - 1)
```

Memoization

Execution of the call *fibonacci(5)*

How many times does it call *fibonacci(1)*?



Memoization

- We can use a dictionary to count functions calls in the program's execution.
  - A dictionary can creates an arbitrary amount of new storage during a program's execution.

Codes

```
# This dictionary will keep track of the number of calls to the
# fibonacci function.
call_counter = {}

def fibonacci(n):
    """ Returns the nth Fibonacci number. """
    # Count the call
    if n not in call_counter:
        call_counter[n] = 1
    else:
        call_counter[n] += 1
```

Codes

```
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 2) + fibonacci(n - 1)


# Call fibonacci(5)
print("fibonacci(5) = ", fibonacci(5))
print()

# Report the total number of calls to the fibonacci function
print("Argument    Calls")
print("----------------")
for args, calls in sorted(call_counter.items()):
    print("   ", args, "         ", calls);
```

Output

```
fibonacci(5) =  5

Argument    Calls
----------------
   0          3
   1          5
   2          3
   3          2
   4          1
   5          1
```

- It shows the number of function calls for, *fibonacci(5), fibonacci(4), …, fibonacci(0).*
- The algorithm is not scalable.
  - An invocation of *fibonacci(35)* will calls *fibonacci(1)* over one million times.

Memoization

- We can improve the performance of our *fibonacci* function using a technique known as **memoization**, which records the result of a specific computation so that the result can be used as needed later during the algorithm's execution.

Codes

```
from time import clock
from random import randrange


def fibonacci(n):
    """ Returns the nth Fibonacci number recursively. """
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 2) + fibonacci(n - 1)
```

Codes

```
# Dictionary for caching the results of the fibonacci2 function
ans = {0:0, 1:1}  # Precompute the results for 0 and 1

def fibonacci2(n):
    """ Returns the nth Fibonacci number.  Caches a
        recursively computed result to be used when needed
        in the future.  Provides a huge performance improvement
        over the recursive version.  """
    if n not in ans.keys():
        result = fibonacci2(n - 2) + fibonacci2(n - 1)
        ans[n] = result
    return ans[n]


def time_it(f, ns):
    """ f is a function that accepts a single parameter.
        ns is a list.
        Measures the time for function f to process each element in ns.
        Returns the cumulative elapsed time. """
    start_time = clock()
    for i in ns:
```

Codes

```
            #print("{:>4}: {:>8}".format(i, f(i)))
            print(f(i), end=" ")
    end_time = clock()
    return end_time - start_time  #  Return elapsed time

def main():
    """ Tests the performance of the fibonacci and fibonacci2 functions. """

    # Make a list of pseudorandom integers in the range 1...50
    numbers = []
    for i in range(10):
        numbers.append(randrange(40) + 1)

    # Print the numbers
    print(numbers)

    # Compare the two Fibonacci functions
    print("Time:", time_it(fibonacci, numbers))
    print("------------------")
    print("Time:", time_it(fibonacci2, numbers))
if __name__ == "__main__":
    main()
```

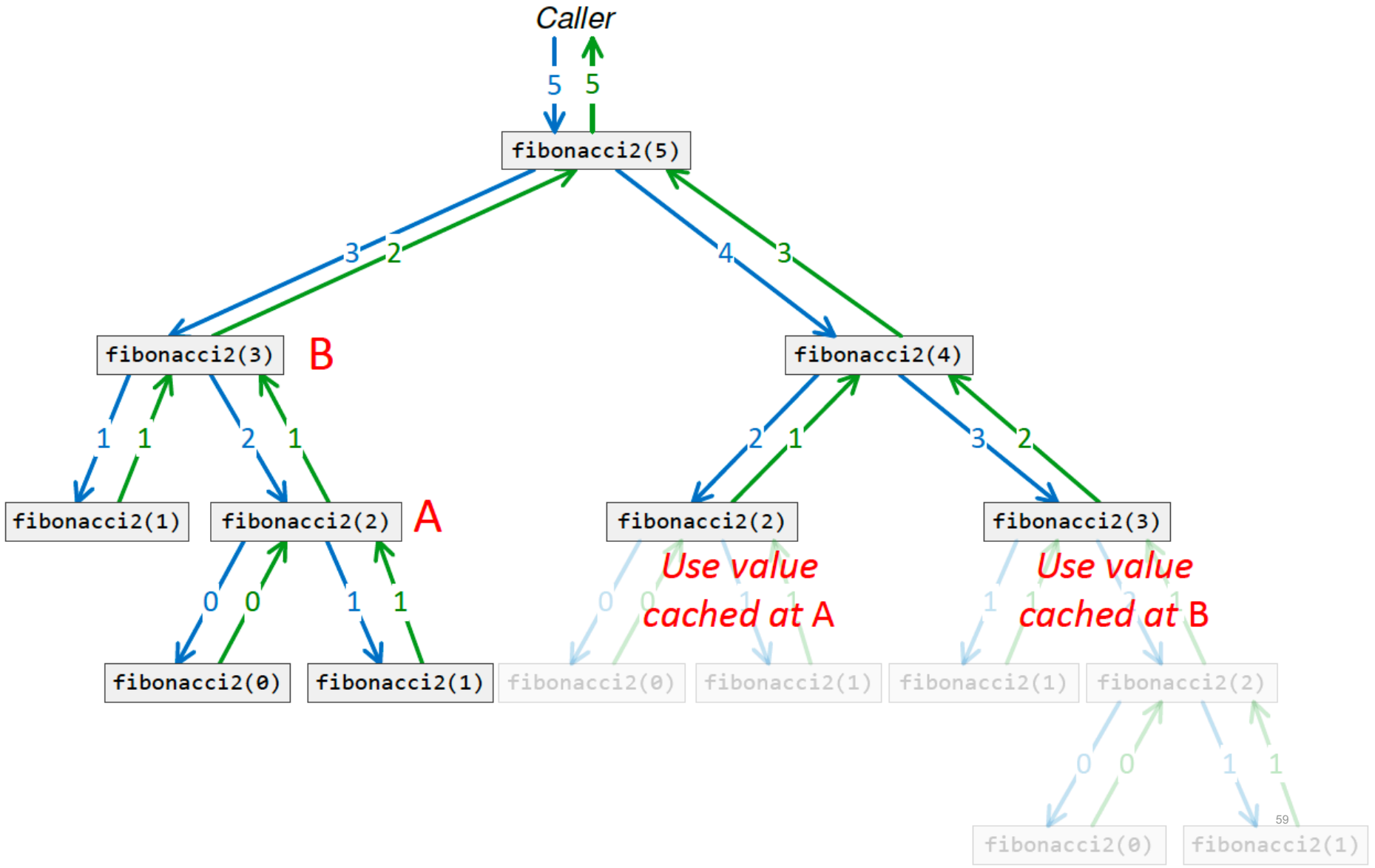
Output

```
[23, 40, 32, 7, 8, 1, 29, 30, 10, 36]
28657 102334155 2178309 13 21 1 514229 832040 55 14930352
Time:  80.61978684888133
------------------
28657 102334155 2178309 13 21 1 514229 832040 55 14930352
Time:  0.00012145362886428757
```

- The non-memoized version requires about 80 seconds to compute the Fibonacci numbers for a list of 10 relatively small values, while the memoized version takes about 0.00012 second for the same list of values.
- The memoized version is over 600,000 times faster than the non-memoized version.

Memoization

Execution of the call *fibonacci2(5)*



Memoization

- A sequence is an ordered list of elements, such as a Python list, tuple, or string.
- A subsequence is a sequence formed from another sequence by removing elements without changing the relative order of the remaining elements.
  - For example, the string "ACD" is a subsequence of "ABCD".
- The longest common subsequence (LCS) problem: given two sequences of symbols, find the longest subsequence that is common to both sequences.
  - This problem is related to genome sequencing problems in computational biology.

Memoization

- The LCS problem has a recursive solution.

```
def LCS(X, Y):
    """ Computes the longest common subsequence of strings X and Y. """

    result = ''  # No symbols in common unless determined otherwise
    if len(X) > 0 and len(Y) > 0:  #  No symbols in common if either string is empty
        Xrest = X[1:]   #  String X without its first symbol
        Yrest = Y[1:]   #  String Y without its first symbol
        if X[0] == Y[0]:  # Do the first symbols of both strings match?
            # An LCS will include this shared symbol plus the LCS of rest
            # of both strings
            result = X[0] + LCS(Xrest, Yrest)
        else:
            # Compare string X to the string Y, excluding Y's first symbol
            X_Y1 = LCS(X, Yrest)
            # Compare string X, excluding X's first symbol, to string Y
            X1_Y = LCS(Xrest, Y)
            #  Choose longer of the two computed sequences
            result = X_Y1 if len(X_Y1) > len(X1_Y) else X1_Y
    return result
```

```
from stopwatch import Stopwatch # Our Stopwatch class from before
from random import choice
def LCS(X, Y):
    """ Computes the longest common subsequence of strings X and Y. """

    result = ''  # No symbols in common unless determined otherwise
    if len(X) > 0 and len(Y) > 0:  #  No symbols in common if either string is empty
        Xrest = X[1:]   #  String X without its first symbol
        Yrest = Y[1:]   #  String Y without its first symbol
        if X[0] == Y[0]:  # Do the first symbols of both strings match?
            # An LCS will include this shared symbol plus the LCS of rest
            # of both strings
            result = X[0] + LCS(Xrest, Yrest)
        else:
            # Compare string X to the string Y, excluding Y's first symbol
            X_Y1 = LCS(X, Yrest)
            # Compare string X, excluding X's first symbol, to string Y
            X1_Y = LCS(Xrest, Y)
            #  Choose longer of the two computed sequences
            result = X_Y1 if len(X_Y1) > len(X1_Y) else X1_Y
    return result
```

```
def LCS_memoized(X, Y):
    """ Computes the longest common subsequence of strings X and Y.
        Caches intermediate results in a memoization dictionary so they
        do not need to be recomputed over and over again. """

    memo = {}   # Start with empty memoization storage

    def LCS(X, Y):
        """ Computes the longest common subsequence of strings X and Y. """
        nonlocal memo  # Memoization storage for computed results
        # Check first to see if we already have computed
        if (X, Y) in memo:
            result = memo[X, Y]   # LCS of X and Y already computed, so use previous value
        else:
            result = ''   # No symbols in common unless determined otherwise
            if len(X) > 0 and len(Y) > 0:  #  No symbols in common if either string is empty
                Xrest = X[1:]   #  String X without its first symbol
                Yrest = Y[1:]   #  String Y without its first symbol
                if X[0] == Y[0]:  # First symbols of both strings match
                    result = X[0] + LCS(Xrest, Yrest)
                else:
```

```
                    X_Y1 = LCS(X, Yrest)
                    X1_Y = LCS(Xrest, Y)
                    #  Choose longest
                    result = X_Y1 if len(X_Y1) > len(X1_Y) else X1_Y
            # Remember this result for X and Y to avoid recomputing it in the future
            memo[X, Y] = result
        return result

    result =  LCS(X, Y)
    print("******** Stored memos:", len(memo))
    return result


def highlight(seq1, seq2):
    """ Highlights the characters of seq2 within seq1. """
    pos1, pos2 = 0, 0
    print(seq1)  # Print the string on one line
    while pos1 < len(seq1):
        if pos2 < len(seq2) and seq1[pos1] == seq2[pos2]:
            print('^', end='')  # Print "^" if characters match
            pos2 += 1
```

```
        else:
            print(' ', end='')  # Print a space to move to the next position
        pos1 += 1
    print()


def compare_LCS(seq1, seq2):
    """ Computes the longest common subsequence of seq 1 and seq2 in
        two different ways and compares the execution times of the two
        approaches. """
    timer_std = Stopwatch()     # Each function has its own stopwatch
    timer_memo = Stopwatch()    # to keep track of elapsed time independently
    timer_std.start()           # Time the standard LCS function
    subseq_std = LCS(seq1, seq2)
    timer_std.stop()
    timer_memo.start()          # Time the memoized LCS function
    subseq_memo = LCS_memoized(seq1, seq2)
    timer_memo.stop()
    # Report results
    print(seq1)
    print(seq2)
    print(subseq_std, len(subseq_std), '(Standard: {:f})'.format(timer_std.elapsed()))
```

```
    print(subseq_memo, len(subseq_memo), '(Memoized: {:f})'.format(timer_memo.elapsed()))
    print()
    # Show the computed longest common subsequences
    highlight(seq1, subseq_std)
    highlight(seq2, subseq_std)
    print()
    highlight(seq1, subseq_memo)
    highlight(seq2, subseq_memo)

    return timer_std.elapsed(), timer_memo.elapsed()

def build_random_string(n, symbols):
    """ Builds a string of length n with characters selected
        pseudorandomly from the string parameter symbols. """
    result = ""
    while len(result) < n:
        result = result + choice(symbols)  # Add a random character from symbols
    return result
```

```
def main():
    N = 10            # Number of times to perform a series of experiments
    max_length = 18   # Maximum string length for experiments

    #  Make a list of tuples for measuring the time to compute the
    #  LCS two different ways; initialize all the times to zero
    data = [(0, 0) for _ in range(max_length)]

    # Perform the series of experiments N times to produce
    # more reliable results
    for runs in range(N):
        # The series of experiments consists of computing the LCS
        # of two strings of increasing length
        for i in range(max_length):
            print(">>> Run", runs, "  String length", i)
            #  Construct two random DNA base sequences
            seq1 = build_random_string(i, "ACGT")
            seq2 = build_random_string(i, "ACGT")

            # Perform an experiment to time the two LCS functions
            lcs_time, memo_time = compare_LCS(seq1, seq2)
```

```
            # Accumulate the times
            data[i] = data[i][0] + lcs_time, data[i][1] + memo_time

    print(data)

if __name__ == '__main__':
    main()
```

Output of *print(data)*:

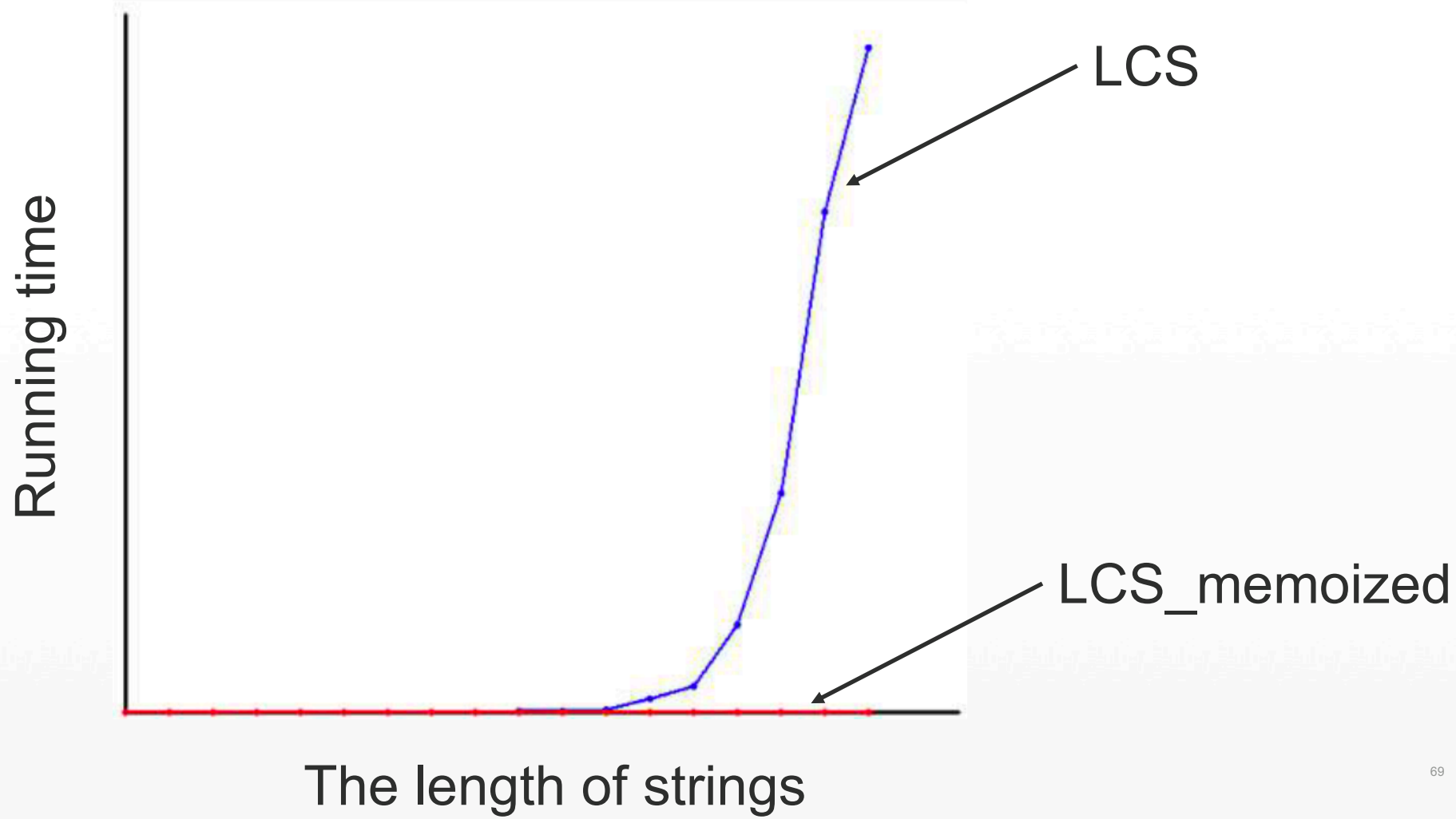
```
[(8.999999999914632e-05, 0.0004280000000131956), (4.599999999010951e-05, 0.0009469999999930367), (8.699999998651453e-
05, 0.0008330000000243976), (0.0001770000001463767, 0.0007589999999965125), (0.0002180000000119308 7, 0.0005660000000
007326), (0.0004909999999989139, 0.0009729999999938954), (0.001496000000046048, 0.0010249999999657478), (0.0041740000
00004119, 0.0024580000000150592), (0.007532000000005867, 0.002360000000016238), (0.01143799999999262, 0.0032779999999
866805), (0.040391999999999876, 0.0036230000000082585), (0.14481300000000963, 0.0015030000000085808), (0.3845610000000
115, 0.0081229999999993108), (1.1179960000000195, 0.0028490000000056526), (3.4620339999999867, 0.0025499999999873957),
(13.849766999999973, 0.0027420000000271116), (15.757560000000012, 0.0031969999999997896), (88.49873900000001, 0.004125
000000032131)]
```

Time of LCS

Time of LCS_memoized

Memoization

- Running time with respect to the length of strings.



References

- *Think Python: How to Think Like a Computer Scientist*, 2nd edition, 2015, by Allen Downey
- *Python Cookbook*, 3rd edition, by David Beazley and Brian K. Jones, 2013
- *Python for Data Analysis*, 2nd edition, by Wes McKinney, 2018
- *Fundamentals of Python Programming,* by Richard L. Halterman