Out[1]:

Chapter 2: Lists

Chapter 2: Lists

From Theory to Python

List performance

Table from the book Chapter 2.6: Lists (http://interactivepython.org/runestone/static/pythonds/AlgorithmAnalysis/Lists.html)

Operation	Big-O Efficiency
index []	O(1)
index assignment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,item)	O(n)
del operator	O(n)
iteration	O(n)
contains (in)	O(n)
get slice [x:y]	O(k)
del slice	O(n)
set slice	O(n+k)
reverse	O(n)
concatenate	O(k)
sort	O(n log n)
multiply	O(nk)

Fast or not?

```
x = ["a", "b", "c"]

x[2]        x[2] = "d"       x.append("d")     x.insert(0, "d")        x[3:5]        x.s
ort()
```

What about `len(x)` ? If you don't know the answer, try googling it!

Sublist iteration performance

`get slice` time complexity is `O(k)`, but what about memory? It's the same!

So if you want to iterate a part of a list, beware of slicing! For example, slicing a list like this can occupy much more memory than necessary:

In [2]:

```
x = range(1000)

print [2*y for y in x[100:200]]
```

```
[200, 202, 204, 206, 208, 210, 212, 214, 216, 218, 220, 222, 224, 226, 228, 230, 2
32, 234, 236, 238, 240, 242, 244, 246, 248, 250, 252, 254, 256, 258, 260, 262, 264
, 266, 268, 270, 272, 274, 276, 278, 280, 282, 284, 286, 288, 290, 292, 294, 296,
298, 300, 302, 304, 306, 308, 310, 312, 314, 316, 318, 320, 322, 324, 326, 328, 33
0, 332, 334, 336, 338, 340, 342, 344, 346, 348, 350, 352, 354, 356, 358, 360, 362,
 364, 366, 368, 370, 372, 374, 376, 378, 380, 382, 384, 386, 388, 390, 392, 394, 3
96, 398]
```

The reason is that, depending on the Python interpreter you have, slicing like `x[100:200]`at loop start can create a *new* list. If we want to explicitly tell Python we just want to iterate through the list, we can use the so called itertools (https://docs.python.org/2/library/itertools.html). In particular, the islice (https://docs.python.org/2/library/itertools.html#itertools.islice) method is handy, with it we can rewrite the list comprehension above like this:

In [3]:

```
import itertools

print [2*y for y in itertools.islice(x, 100, 200)]
```

```
[200, 202, 204, 206, 208, 210, 212, 214, 216, 218, 220, 222, 224, 226, 228, 230, 2
32, 234, 236, 238, 240, 242, 244, 246, 248, 250, 252, 254, 256, 258, 260, 262, 264
, 266, 268, 270, 272, 274, 276, 278, 280, 282, 284, 286, 288, 290, 292, 294, 296,
298, 300, 302, 304, 306, 308, 310, 312, 314, 316, 318, 320, 322, 324, 326, 328, 33
0, 332, 334, 336, 338, 340, 342, 344, 346, 348, 350, 352, 354, 356, 358, 360, 362,
 364, 366, 368, 370, 372, 374, 376, 378, 380, 382, 384, 386, 388, 390, 392, 394, 3
96, 398]
```

Exercises

Implement `swap`

Try to code and test the `swap` function from selection sort (slide 29 theory) (http://disi.unitn.it/~montreso/sp/slides/03-analisi.pdf):

$$
\begin{array}{l}
\mathsf{swap}(\textsc{Item}[\,]\ A, \textbf{int}\ x, \textbf{int}\ y) \\
\quad \textbf{int}\ temp = A[x] \\
\quad A[x] = A[y] \\
\quad A[y] = temp
\end{array}
$$

- Use the following skeleton to code it
- Check carefully all the test cases, in particular `test_swap_property` and `test_double_swap`. They show two important properties of the swap function. Make sure you understand why these tests should succeed.

In [4]:

```python
import unittest

def swap(A, x, y):
    """
    In the array A, swaps the elements at position x and y.
    """
    raise Exception("TODO implement me!")

class SwapTest(unittest.TestCase):

    def test_one_element(self):
        v = ['a'];
        swap(v,0,0)
        self.assertEqual(v, ['a'])

    def test_two_elements(self):
        v = ['a','b'];
        swap(v,0,1)
        self.assertEqual(v, ['b','a'])

    def test_return_none(self):
        v = ['a','b', 'c', 'd'];
        self.assertEquals(None, swap(v,1,3))


    def test_long_list(self):
        v = ['a','b', 'c', 'd'];
        swap(v,1,3)
        self.assertEqual(v, ['a', 'd','c', 'b'])


    def test_swap_property(self):
        v = ['a','b','c','d'];
        w = ['a','b','c','d'];
        swap(v,1,3)
        swap(w,3,1)
        self.assertEqual(v, w)

    def test_double_swap(self):
        v = ['a','b','c','d'];
        swap(v,1,3)
        swap(v,1,3)
        self.assertEqual(v, ['a','b','c','d'])
```

Implement partial_min_pos

Try to code and test the partial `min` pos function from selection sort (slide 29 theory) (http://disi.unitn.it/~montreso/sp/slides/03-analisi.pdf):

```
int min(ITEM[] A, int i, int n)
    int min = i                          % Partial minimum position
    for j = i + 1 to n − 1 do
        if A[j] < A[min] then
            min = j                      % New partial minimum
    return min
```

- Use the following skeleton to code it
- add some test to the provided testcase class

Notice that

- we renamed `min` to `partial_min_pos` to avoid name collision with Python standard library `min` function
- it is not necessary to pass list length n, as it is already stored in Python implementation of lists, and we can retrieve it in `O(1)` time with `len(A)`

In [5]:

```python
import unittest

def partial_min_pos(A, i):
    """
    Return the index of the element in list A which is lesser or equal than all other values in A
    that start from index i included
    """
    raise Exception("TODO implement me!")

class PartialMinPosTest(unittest.TestCase):

    def test_one_element(self):
        self.assertEqual(partial_min_pos([1],0),0)

    def test_two_elements(self):
        self.assertEqual(partial_min_pos([1,2],0),0)
        self.assertEqual(partial_min_pos([2,1],0),1)
        self.assertEqual(partial_min_pos([2,1],1),1)

    def test_long_list(self):
        self.assertEqual(partial_min_pos([8,9,6,5,7],2),3)
```

Implement `selection_sort`

Try to code and test the `selectionSort` from selection sort (slide 29 theory) (http://disi.unitn.it/~montreso/sp/slides/03-analisi.pdf):

$$\mathsf{selectionSort}(\textsc{Item}[\,]\ A, \textbf{int}\ n)$$

$$\begin{array}{l} \textbf{for}\ i = 0\ \textbf{to}\ n-2\ \textbf{do} \\ \quad \textbf{int}\ min = \mathsf{min}(A, i, n) \\ \quad \mathsf{swap}(A, i, min) \end{array}$$

Use the following skeleton to code it and add some test to the provided testcase class.

Notice that

- we renamed `selectionSort` to `selection_sort` because it is a more pythonic name (https://www.python.org/dev/peps/pep-0008/#function-names)
- it is not necessary to pass list length `n`, as it is already stored in Python implementation of lists, and we can retrieve it in `O(1)` time with `len(A)`
- On the book website, there is an implementation of the selection sort (http://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html) with a nice animated histogram showing a sorting process. Differently from the slides, instead of selecting the minimal element the algorithm on the book selects the *maximal* element and puts it to the right of the array.

In [6]:

```
import unittest

def selection_sort(A):
    """
    Sorts the list A in-place in O(n^2) time.
    """
    raise Exception("TODO implement me!")

class SelectionSortTest(unittest.TestCase):

    def test_zero_elements(self):
        v = []
        selection_sort(v)
        self.assertEqual(v,[])

    def test_return_none(self):
        self.assertEquals(None, selection_sort([2]))

    def test_one_element(self):
        v = ["a"]
        selection_sort(v)
        self.assertEqual(v,["a"])


    def test_two_elements(self):
        v = [2,1]
        selection_sort(v)
        self.assertEqual(v,[1,2])

    def test_three_elements(self):
        v = [2,1,3]
        selection_sort(v)
        self.assertEqual(v,[1,2,3])

    def test_piccinno_list(self):
        v = [23, 34, 55, 32, 7777, 98, 3, 2, 1]
        selection_sort(v)
        vcopy = v[:]
        vcopy.sort()
        self.assertEqual(v, vcopy)
```

Implement gap_rec

Try to code and test the `gap` function from recursion theory slides (slide 21) (http://disi.unitn.it/~montreso/sp/slides/02-recursion.pdf):

```
gap(int[] L, int i, int j)
--------------------------
if j == i + 1 then
  return j
m = ⌊(i + j)/2⌋
if L[m] ≤ L[i] then
  return gap(L, m, j)
else
  return gap(L, i, m)
```

Use the following skeleton to code it and add some test to the provided testcase class.To understand what's going on, try copy pasting your solution in Python tutor (http://pythontutor.com/visualize.html#mode=edit) and hit `Visualize execution` and then `Forward` to step through the process

Notice that

- We created a function `gap_rec` to differentiate it from the iterative one
- Users of `gap_rec` function might want to call it by passing just a list, in order to find any gap in the whole list. So for convenience the new function `gap_rec(L)` only accepts a list, without indexes `i` and `j`. This function just calls the other function `gap_rec_helper` that will actually contain the recursive calls. So your task is to translate the pseudocode of `gap` into the Python code of `gap_rec_helper`, which takes as input the array and the indexes as `gap` does. Adding a helper function is a frequent pattern you can find when programming recursive functions.

> **WARNING: The specification of gap_rec assumes the input is always a list of at least two elements, and that the first element is less or equal than the last one.**
> ***If these conditions are not met, function behaviour could be completely erroneus!***

When preconditions are not met, execution could stop because of an error like index out of bounds, or, even worse, we might get back some wrong index as a gap! To prevent misuse of the function, a good idea can be putting a check at the beginning of the `gap_rec` function. Such check should immediately stop the execution and raise an error if the parameters don't satisfy the preconditions. One way to do this could be to write some assertion (testing#Assertions) like this:

```
assert len(L) >= 2
    assert L[0] <= L[len(L)-1]
```

- These commands will make python interrupt execution and throw an error as soon it detects list `L` is too small or with wrong values
- This kind of behaviour is also called *fail fast*, which is better than returning wrong values!
- You can put any condition you want after `assert`, but ideally they should be fast to execute.

In [8]:

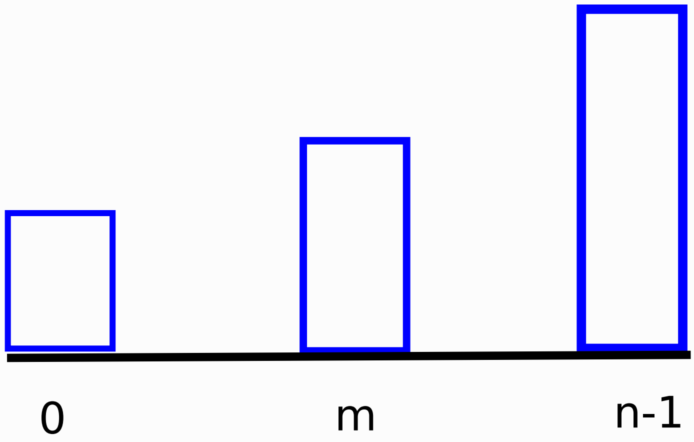
```

```

> **GOOD PRACTICE: Notice I wrote as a comment what the helper function is expected to receive. Writing down specs often helps understanding what the function is supposed to do, and helps users of your code as well!**

> **COMMANDMENT: You shall also write on paper!**

To get an idea of how `gap_rec` is working, draw histograms on paper like the following, with different heights at index `m`:



Notice how at each recursive call, we end up with a histogram that is similar to the inital one, that is, it respects the same preconditions (a list of size >= 2 where first element is smaller or equal than the last one)

- Look at the iterative gap here:

gap(**int**[] L, **int** n)

for $i = 1$ **to** $n - 1$ **do**
    if $L[i-1] < L[i]$ **then**
        return i

return -1

```python
def gap_iter(L):
    for i in range(1,len(L)):
        if L[i-1] < L[i]:
            return i
    return -1
```

- What is the complexity of `gap_rec`? Is it faster or slower than `gap_iter` ?
- Assuming L contains `n >= 2` integers such that `L[0] < L[n-1]`, will the recursive gap always give the same result as the iterative one? If we just change function names, can we run the same test case against both implementations? (Careful!)

In [10]:

```
import unittest

def gap_rec(L):
    """ Searches a gap in list L

    Given a list L containing n >= 2 integers such that L[0] < L[n-1], returns a gap in the li
st.
    A gap is an index i, 0 < i < n such that L[i-1] < L[i]
    """
    return gap_helper(L, 0, len(L)-1)

def gap_helper(L, i, j):
    """ Searches a gap in sublist L[i:j]

    Given a list L containing n >= 2 integers such that L[i] < L[j], returns a gap in the subl
ist L[i:j]
    A gap is an index z, i < z < j+1 such that L[z-1] < L[z]
    """
    raise Exception("TODO implement me!")

class GapRecTest(unittest.TestCase):

    def test_two_elements(self):
        self.assertEqual(gap_rec([1,2]),1)

    def test_three_elements_middle(self):
        self.assertEquals(gap_rec([1,3,3]), 1)

    def test_three_elements_right(self):
        self.assertEquals(gap_rec([1,1,3]), 2)
```

Implement binary_search_rec

Try to code and test the `binarySearch` recursive function from recursion theory slides (slide 21) (http://disi.unitn.it/~montreso/sp/slides/02-recursion.pdf):

```
int binarySearch(int[] L, int v, int i, int j)

if i > j then
    return −1
else
    int m = ⌊(i + j)/2⌋
    if L[m] = v then
        return m
    else if L[m] < v then
        return binarySearch(L, v, m + 1, j)
    else
        return binarySearch(L, v, i, m − 1)
```

- Use the following skeleton to code it
- add some test to the provided testcase class
- Does the pseudocode algorithm work with the empty list?
- What happens if we allow non-distinct numbers? Does it work anyway?
- What is the time complexity of the recursive version?
- What is the memory complexity of the recursive version?

Notice that

- we renamed `binarySearch` to `binary_search_rec` to have more pythonic name and differentiate it from the iterative one
- the renamed function `binary_search_rec(L)` only accepts a list, without indexes `i` and `j`, we will need a way to specify them when we translate the pseudocode. You can follow the same pattern used for `gap_rec_helper`
    - SUGGESTION : write as a comment what the helper function is expected to receive. Can it receive an empty list? Can it receive indices out of bounds? You decide the assumptions, but once they are decided you should make sure that unacceptable values don't get into it!
- To understand what's going on, try copy pasting your solution in Python tutor (http://pythontutor.com/visualize.html#mode=edit) and hit `Visualize execution` and then Forward to step through the process
- Remember that even experienced programmers tend to fail implementing the binary search at first time, it's easy to get wrong indexes! So good tests here can really spot issues.

In [11]:

```python
import unittest

def binary_search_rec(L,v ):
    """ Searches value v in sorted list L

    Given a list L containing n distinct sorted integers, returns the index position
    of element with value v, or -1 if not found
    """

    raise Exception("TODO implement me!")

class BinarySearchRecTest(unittest.TestCase):

    def test_empty(self):
        self.assertEqual(binary_search_rec([], 7), -1)

    def test_one_element_found(self):
        self.assertEqual(binary_search_rec([7],7),0)

    def test_one_element_not_found(self):
        self.assertEqual(binary_search_rec([6],7),-1)

    def test_one_negative_element_not_found(self):
        self.assertEqual(binary_search_rec([-7],7),-1)

    def test_two_elements_found_right(self):
        self.assertEquals(binary_search_rec([6,7],7), 1)

    def test_two_elements_not_found(self):
        self.assertEquals(binary_search_rec([6,7],3), -1)

    def test_two_elements_found_left(self):
        self.assertEquals(binary_search_rec([6,7],6), 0)

    def test_long_list(self):
        self.assertEquals(binary_search_rec([2,4,5,7,9],7), 3)
```

Implement binary_search_iter

Try to code and test the `iterativeBinarySearch` function from Introduction slides (slide 18) (http://disi.unitn.it/~montreso/sp/slides/01-introduzione.pdf):

int iterativeBinarySearch(**int**[] A, **int** n, **int** v)

int $i = 0$
int $j = n - 1$
int $m = \lfloor (i + j)/2 \rfloor$
while $i < j$ **and** $A[m] \neq v$ **do**
  if $A[m] < v$ **then**
    $i = m + 1$
  else
    $j = m - 1$
  $m = \lfloor (i + j)/2 \rfloor$
if $i > j$ **or** $A[m] \neq v$ **then**
  return -1
else
  return m

- This time, there's no code skeleton and you're on your own!
- Try to reuse test cases from the recursive version
- What is the time complexity of the iterative version? Is it different from the recursive version?
- What is the memory complexity of the iterative version? Is it different from the recursive version?

Solutions

`swap` solution

In [12]:

```
import unittest

def swap(A, x, y):
    """
    In the array A, swaps the elements at position x and y.
    """
    temp = A[x]
    A[x] = A[y]
    A[y] = temp

class SwapTest(unittest.TestCase):

    def test_one_element(self):
        v = ['a'];
        swap(v,0,0)
        self.assertEqual(v, ['a'])

    def test_two_elements(self):
        v = ['a','b'];
        swap(v,0,1)
        self.assertEqual(v, ['b','a'])

    def test_return_none(self):
        v = ['a','b', 'c', 'd'];
        self.assertEquals(None, swap(v,1,3))

    def test_long_list(self):
        v = ['a','b', 'c', 'd'];
        swap(v,1,3)
        self.assertEqual(v, ['a', 'd','c', 'b'])


    def test_swap_property(self):
        v = ['a','b','c','d'];
        w = ['a','b','c','d'];
        swap(v,1,3)
        swap(w,3,1)
        self.assertEqual(v, w)

    def test_double_swap(self):
        v = ['a','b','c','d'];
        swap(v,1,3)
        swap(v,1,3)
        self.assertEqual(v, ['a','b','c','d'])
```

`partial_min_pos` solution

In [14]:

```
import unittest

def partial_min_pos(A, i):
    """
    Return the index of the element in list A which is lesser or equal than all other values i
n A
    that start from index i included
    """
    pm = i

    for j in range(i+1, len(A)):
        if (A[j] < A[pm]):
            pm = j
    return pm

class PartialMinPosTest(unittest.TestCase):

    def test_one_element(self):
        self.assertEqual(partial_min_pos([1],0),0)

    def test_two_elements(self):
        self.assertEqual(partial_min_pos([1,2],0),0)
        self.assertEqual(partial_min_pos([2,1],0),1)
        self.assertEqual(partial_min_pos([2,1],1),1)

    def test_long_list(self):
        self.assertEqual(partial_min_pos([8,9,6,5,7],2),3)
```

`selection_sort` solution

In [16]:

```
import unittest

def selection_sort(A):
    """
    Sorts the list A in-place in O(n^2) time.
    """
    for i in range(0, len(A)-1):
        m = partial_min_pos(A, i)
        swap(A, i, m)

class SelectionSortTest(unittest.TestCase):

    def test_zero_elements(self):
        v = []
        selection_sort(v)
        self.assertEqual(v,[])

    def test_return_none(self):
        self.assertEquals(None, selection_sort([2]))

    def test_one_element(self):
        v = ["a"]
        selection_sort(v)
        self.assertEqual(v,["a"])


    def test_two_elements(self):
        v = [2,1]
        selection_sort(v)
        self.assertEqual(v,[1,2])

    def test_three_elements(self):
        v = [2,1,3]
        selection_sort(v)
        self.assertEqual(v,[1,2,3])

    def test_piccinno_list(self):
        v = [23, 34, 55, 32, 7777, 98, 3, 2, 1]
        selection_sort(v)
        vcopy = v[:]
        vcopy.sort()
        self.assertEqual(v, vcopy)
```

gap_rec solution

In [18]:

```
import unittest

def gap_rec(L):
    """
    Given a list L containing n >= 2 integers such that L[0] < L[n-1], returns a gap in the li
st.
    A gap is an index i, 0 < i < n such that L[i-1] < L[i]
    """
    return gap_helper(L, 0, len(L)-1)

def gap_helper(L, i, j):
    """
    Given a list L containing n >= 2 integers such that L[i] < L[j], returns a gap in the subl
ist L[i:j]
    A gap is an index z, i < z < j+1 such that L[z-1] < L[z]
    """
    if j == i + 1:
        return j
    m = (i+j) // 2   # remember in every python version // operator behaves the same and floor
s the result

    if (L[m] <= L[i]):
        return gap_helper(L, m, j)
    else:
        return gap_helper(L, i, m)

class GapRecTest(unittest.TestCase):

    def test_two_elements(self):
        self.assertEqual(gap_rec([1,2]),1)

    def test_three_elements_middle(self):
        self.assertEquals(gap_rec([1,3,3]), 1)

    def test_three_elements_right(self):
        self.assertEquals(gap_rec([1,1,3]), 2)
```

binary_search_rec solution

In [20]:

```
import unittest

def binary_search_rec(L,v ):
    """ Searches value v in sorted list L

    Given a list L containing n distinct sorted integers, returns the index position
    of element with value v, or -1 if not found
    """

    return binary_search_helper(L,v, 0, len(L)-1)

def binary_search_helper(L, v, i, j):
    """ Helper for the recursive binary search

    Given a list L containing n distinct sorted integers, returns the index position
    of element with value v if it is present in sublist L[i:j], or -1 if not found
    """
    if i > j:
        return -1

    m = (i+j) // 2   # remember in every python version // operator behaves the same and floor
s the result

    # print "L = ", L
    # print "v = ", v
    # print "m = ", m
    # print "i = ", i
```

```python
    # print "j = ", j


    if L[m] == v:
        return m
    elif L[m] < v:
        return binary_search_helper(L, v, m + 1, j)
    else:
        return binary_search_helper(L, v, i, m - 1)

class BinarySearchRecTest(unittest.TestCase):

    def test_empty(self):
        self.assertEqual(binary_search_rec([], 7), -1)

    def test_one_element_found(self):
        self.assertEqual(binary_search_rec([7],7),0)

    def test_one_element_not_found(self):
        self.assertEqual(binary_search_rec([6],7),-1)

    def test_one_negative_element_not_found(self):
        self.assertEqual(binary_search_rec([-7],7),-1)

    def test_two_elements_found_right(self):
        self.assertEquals(binary_search_rec([6,7],7), 1)

    def test_two_elements_not_found(self):
        self.assertEquals(binary_search_rec([6,7],3), -1)

    def test_two_elements_found_left(self):
        self.assertEquals(binary_search_rec([6,7],6), 0)

    def test_long_list(self):
        self.assertEquals(binary_search_rec([2,4,5,7,9],7), 3)

    def test_not_distinct_found(self):
        self.assertEquals(binary_search_rec([7,7],7), 0)

    def test_not_distinct_not_found(self):
        self.assertEquals(binary_search_rec([7,7],5), -1)
```