Lab 6

Sorting

Sup'Biotech 3

Python

Pierre Parutto

November 23, 2016

Preamble

Document Property

Authors	Pierre Parutto
Version	1.0
Number of pages	11

Contact

Contact the assistant team at: supbiotech-bioinfo-bt3@googlegroups.com

Copyright

Contents

1 Introduction

In this lab we will investigate different algorithms to sort values in a list. We will restrict ourselves to lists containing only integers but all the presented algorithms also work on any data type for which comparison is defined.

2 Algorithms For Sorting

First we are going to study algorithms for sorting an already existing list.

2.1 Insertion Sort

The insertion sort is the most straightforward sorting algorithm: given a list, it traverses the list, inserting at each turn the minimal element in a new list.

This algorithm does not modify the input list.

1. Formalize the problem: Write the steps to perform (pseudo-code)
2. Write the function `insertion_sort(l: list) -> list`

Example

```
>>> insertion_sort([8, 6, 3, 5, 4, 1, -5])
[-5, 1, 3, 4, 5, 6, 8]
```

Correction:

```
def insertion_sort(l):
    if len(l) <= 1:
        return l

    res = []
    done = []
    for i in range(len(l)):
        done.append(False)

    for i in range(len(l)):
        cur_min = 999
        pos_min = -1
        for j in range(len(l)):
            if done[j] == False and l[j] < cur_min:
                cur_min = l[j]
                pos_min = j
        done[pos_min] = True
        res.append(l[pos_min])
    return res
```

2.2 Selection Sort

The selection sort algorithm is very similar to the insertion sort but instead of creating a new list we are going to swap elements. The pseudo-code is the following:

```
input list: l
for i = 1 .. len(l)
    find the position k of the minimal element of l[i+1:]
    swap l[i] and l[k]
return l
```

Write the function `selection_sort(l: list) -> list` that implements the previous pseudo code.

Example

```
>>> selection_sort([8, 6, 3, 5, 4, 1, -5])
[-5, 1, 3, 4, 5, 6, 8]
```

Correction:

```
def selection_sort(l):
    for i in range(len(l)-1):
        min_p = i
        min_v = l[i]

        for k in range(i+1, len(l)):
            if l[k] < min_v:
                min_p = k
                min_v = l[k]

        tmp = l[min_p]
        l[min_p] = l[i]
        l[i] = tmp

    return l
```

2.3 Bubble Sort

The bubble sort algorithm swaps two adjacent elements if the element on the right is smaller than the one on the left.

1. Formalize the problem: Write the steps to perform (pseudo-code)
2. Write the function `bubble_sort(l: list) -> list`

Example

```
>>> bubble_sort([8, 6, 3, 5, 4, 1, -5])
[-5, 1, 3, 4, 5, 6, 8]
```

Correction:

```python
def bubble_sort(l):
    done = False
    n = len(l)
    while not done:
        done = True
        for i in range(1, n):
            if l[i-1] > l[i]:
                tmp = l[i-1]
                l[i-1] = l[i]
                l[i] = tmp
                done = False
        n = n - 1
    return l
```

2.4 Quick Sort

The quick sort algorithm recursively sort the list. Its pseudo-code is the following:

```
input list: l

quick_sort(l):
    if len(l) == 1:
        return l[0]
    choose a pivot element at position p = len(l) / 2
    create the list sub containing only elements less or equal than l[p]
    create the list sup containing only elements greater than l[p]
    return the merged list: quick_sort(sub) + l[p] + quick_sort(sup)
```

Write the function `quick_sort(l: list) -> list` that implements the previous pseudo-code.

Example

```
>>> quick_sort([8, 6, 3, 5, 4, 1, -5])
[-5, 1, 3, 4, 5, 6, 8]
```

Correction:

```python
def quick_sort(l):
    if len(l) == 0:
        return []
    if len(l) == 1:
        return [l[0]]

    p = len(l) // 2
    left = []
    right = []
    for k in range(len(l)):
        if k != p and l[k] < l[p]:
            left.append(l[k])
        elif k != p:
            right.append(l[k])
    return quick_sort(left) + [l[p]] + quick_sort(right)
```

3 Finding An Element in A List

The interest of sorting is to be able to quickly find elements in a list.

3.1 Finding An Element In An Unordered List

In an unordered list, you can stop searching as soon as you have found the element but have to go through the whole list if it does not appear.

Write the function `find_elt_unordered(l: list, e: int) -> bool` that returns `True` if the integer `e` appears in the list `l` and `False` otherwise.

Example

```
>>> find_elt_unordered([8, 6, 3, 5, 4, 1, -5], 4)
True
>>> find_elt_unordered([8, 6, 3, 5, 4, 1, -5], 0)
False
```

Correction:

```
def find_elt_unordered(l, e):
    for ee in l:
        if ee == e:
            return True
    return False
```

3.2 Finding An Element In An Ordered List

In an ordered list you can stop searching as soon as you have found the element or when the current element is greater that the one you are searching.

Write the function `find_elt_ordered(l: list, e: int) -> bool` that returns `True` if the integer `e` appears in the list `l` and `False` otherwise.

Example

```
>>> find_elt_ordered([-5, 1, 3, 4, 5, 6, 8], 4)
True
>>> find_elt_ordered([-5, 1, 3, 4, 5, 6, 8], 0)
```

```
False
>>> find_elt_ordered([-5, 1, 3, 4, 5, 6, 8], 8)
True
>>> find_elt_ordered([-5, 1, 3, 4, 5, 6, 8], 9)
False
```

Correction:

```
def find_elt_ordered(l, e):
    for ee in l:
        if ee == e:
            return True
        if ee > e:
            return False
    return False
```

3.3 Bissection Search Algorithm

The bissection search algorithm is a clever way to search in ordered list. It is somehow similar to the `quick_sort` algorithm. Given a list `l` and an integer `e`, it compares e with the value at the middle of the list, if they are equal it returns true, if `e` < the middle element then it is recursively search in the first half of the list, otherwise on the second half. If the list is empty it returns `False`. Its pseudo-code is the following:

```
input list: l
input integer: e

bissection_search(l, e)
    if l is empty
        return False
    p = len(l/2)
    if e == l[p]
        return True
    else if e < l[p]
        return bissection_search(l[:p], e)
    else
        return bissection_search(l[p+1,:], e)
```

Write the function `bissection_search(l: list, e: int) -> bool` that implements the previous pseudo-code.

Example

```
>>> bissection_search([-5, 1, 3, 4, 5, 6, 8], 4)
True
>>> bissection_search([-5, 1, 3, 4, 5, 6, 8], 0)
False
>>> bissection_search([-5, 1, 3, 4, 5, 6, 8], 8)
True
>>> bissection_search([-5, 1, 3, 4, 5, 6, 8], 9)
False
```

```
>>> bissection_search([-5, 1, 3, 4, 5, 6, 8], -5)
True
>>> bissection_search([-5, 1, 3, 4, 5, 6, 8], -12)
False
```

Correction:

```
def bissection_search(l, e):
    if l == []:
        return False
    if len(l) == 1:
        return e == l[0]

    p = len(l) // 2
    if e == l[p]:
        return True
    if e < l[p]:
        return bissection_search(l[:p], e)
    else:
        return bissection_search(l[p+1:], e)
```

4 Binary Search Trees

A Binary Search Tree (BST) is a data structure that keeps the list of elements sorted. It allows to to quickly find an element in a list.

A BST is a branching data structure where each element possesses exactly two successors and one predecessor. The first successor is itself a BST and thus this data structure is well suited for recursive implementation.

In a BST, we call an element a node, its first successor its left son and the second its right son. A node that has its two successors empty is called a leaf and represented at the bottom of the tree. On the opposite, the node without a predecessor is called the root and is the first element of the structure.

In a BST, for any node, all the elements on its left son must be less than it and all the elements of its right son must be greater than it. Hence the root contains the middle element.

In Python We represent a node as a list of three elements: `[left son, value, right son]` where value is the value associated to the node. An empty tree is represented by the empty list: `[]`.

Example The list: `[1,2,5,7,-5]` is represented by the following BST:

```
[[[], -5, []], 2, [[], 1, []], 5, [[], 7, []]]
```

4.1 BST Insertion

To insert a new value into an existing BST, you have to find the place where it should be and at a new node at this position.

Write the function `BST_insert(t: list, e: int) -> list` that inserts the integer `e` in the BST `t`.

Example

```
>>> BST_insert([[[[], -5, []], 2, [[], 1, []]], 5, [[], 7, []]], 6)
[[[[], -5, []], 2, [[], 1, []]], 5, [[[], 6, []], 7, []]]
>>> BST_insert([[[], 3, []], 4, [[], 5, []]], 1)
[[[[], 1, []], 3, []], 4, [[], 5, []]]
```

Correction:

```
def BST_insert(t, e):
    if t == []:
        return [[], e, []]
    if e < t[1]:
        return([BST_insert(t[0], e), t[1], t[2]])
    if e > t[1]:
        return([t[0], t[1], BST_insert(t[2], e)])
```

4.2 BST Creation

Given a list of integer, it is possible to create the corresponding BST by starting with an empty BST and inserting in it one by one the elements from the list.

Write the function `BST_create(l: list) -> list` that returns the BST corresponding to the list `l`.

Example

```
>>> BST_create([-5, 1, 2, 7, 5])
[[], -5, [[], 1, [[], 2, [[[], 5, []], 7, []]]]]
>>> BST_create([2, 7, -1, 5, -5])
[[[[], -5, []], -1, []], 2, [[[], 5, []], 7, []]]
```

Correction:

```
def BST_create(l):
    BST = []
    for e in l:
        BST = BST_insert(BST, e)
    return BST
```

Questions

- On what depends the created tree?
- What is `bst_create([-5, 1, 2, 5, 7])`?
- What is the problem with this creation method?

Correction:

1. It depends on the order in which the elements are inserted. Basically, the first element inserted in the tree will be the root. If this element is extreme compared to the other values of the list, it will create an unbalanced tree.
2.
```
[[], -5, [[], 1, [[], 2, [[], 5, [[], 7, []]]]]]
```
3. Look at the previous tree, it is a list !! Hence we may loose all the benefits of using BSTs over lists if we are not careful about the shape of the tree. This means that we have to insert the elements in a specific order.

4.3 BST Creation Balanced

We call level of a tree the number of layers it possesses. The balanced BST is the BST possessing the least number of layers possible created from a list.

The balanced BST tree can be easily created from an ordered list through the following recursive procedure: If the list is of size 1 just return a leaf node, otherwise create a node possessing the value of the element at the middle of the list and create its left son through a recursive call on the first half of the list and the right son through a recursive call on the second half of the list. This is very similar to the bissection search though process.

Write the algorithm `BST_create_balanced(l: list) -> list` that implements this algorithm

Example

```
>>> BST_create_balanced([-5, 1, 2, 5, 7])
[[[[], -5, []], 1, []], 2, [[[], 5, []], 7, []]]
>>> BST_create_balanced([-5, 1, 3, 4, 5, 6, 8])
[[[[], -5, []], 1, [[], 3, []]], 4, [[[], 5, []], 6, [[], 8, []]]]
```

Correction:

```
def BST_create_balanced(l):
    if l == []:
        return []
    p = len(l) // 2
    return [BST_create_balanced(l[:p]), l[p], BST_create_balanced(l[p+1:])]
```

4.4 BST Search

To search in a BST, you have to compare the value to be searched with the different nodes until you reach the value or a leaf and starting at the root of the BST. When an empty node is reached it returns False, if the query value is the same as the node value it returns True. On the other hand, if the query value is less than the node value you have to search it in the left son and on the right son if it is greater.

Write the function `BST_search(t: list, e: int) -> list)` that returns True if the integer `l` is in the BST `t` and False otherwise.

Example

```
>>> BST_search([[[[], -5, []], 1, [[], 3, []]], 4, [[[], 5, []], 6, [[], 8, []]]], 1)
True
>>> BST_search([[[[], -5, []], 1, [[], 3, []]], 4, [[[], 5, []], 6, [[], 8, []]]], 2)
False
>>> BST_search([[[[], -5, []], 1, [[], 3, []]], 4, [[[], 5, []], 6, [[], 8, []]]], 8)
True
>>> BST_search([[[[], -5, []], 1, [[], 3, []]], 4, [[[], 5, []], 6, [[], 8, []]]], -5)
True
>>> BST_search([[[[], -5, []], 1, [[], 3, []]], 4, [[[], 5, []], 6, [[], 8, []]]], 10)
False
>>> BST_search([[[[], -5, []], 1, [[], 3, []]], 4, [[[], 5, []], 6, [[], 8, []]]], -123)
False
>>> BST_search([[[], 3, []], 4, [[], 5, []]], 8)
False
>>> BST_search([[[], 3, []], 4, [[], 5, []]], 4)
True
```

Correction:

```
def BST_search(t, e):
    if t == []:
        return False
    if e == t[1]:
        return True
    if e < t[1]:
        return BST_search(t[0], e)
    if e > t[1]:
        return BST_search(t[2], e)
```