Searching. Sorting

Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

December 16, 2015

Overview

About the exam simulation

- You have to complete a problem statement such as those received within 90 minutes
- We've seen a lot of errors that show a lack of practice
- There's really only 1 error in Python - trying to call a method on the wrong type (e.g. getName() on a string)
- Make sure you finish one functionality before working on the next one
- **Run your program often, and from the first minutes of the test**

Feedback for this course

- You can write feedback at **academicinfo.ubbcluj.ro**
- It is both very important for us as well as anonymous
- Write both what you like (so we keep&improve it) and what you don't (so we know what to update)
- Best if you write about all activities (Lecture,seminar and laboratory)

Searching

- Data are available in the internal memory, as a sequence of records $(k_1, k_2, ..., k_n)$
- Search a record having a certain value for one of its fields, called search key.
- If the search is successful, we will have the position of the record in the given sequence.
- When the keys are already sorted we may need to know the insertion place of a new record with this key, such that the sort order is preserved.

Searching - unordered keys

Specification of the searching problem:

Data: $a, n, (k_i, i=0,n-1)$;

Precondition: $n \in N$, $n \geq 0$;

Results: p;

Post-condition: $(0 \leq p \leq n-1$ *and* $a = k_p)$ *or* $(p=-1$ *if key not found).*

We are going to study a few algorithms to solve this problem.

Searching - unordered keys

```
def searchSeq(el,l):
    """
      Search for an element in a list
      el - element
      l - list of elements
      return the position of the element
         or -1 if the element is not in l
    """
    poz = -1
    for i in range(0,len(l)):
        if el==l[i]:
            poz = i
    return poz
```

$$T(n)=\sum_{(i=0)}^{(n-1)} 1=n\in\Theta(n)$$

Searching - unordered keys

```
def searchSucc(el,l):
    """
      Search for an element in a list
      el - element
      l - list of elements
      return the position of first occurrence
             or -1 if the element is not in l
    """
    i = 0
    while i<len(l) and el!=l[i]:
        i=i+1
    if i<len(l):
        return i
    return -1
```

Searching - unordered keys

Best case: the element is at the first position

$T(n) \in \theta(1)$

Worst-case: the element is in the n-1 position

$T(n) \in \theta(n)$

Average case: while can be executed 0,1,2,n-1 times

$T(n) = (1+2+\ldots+n-1)/n \in \Theta(n)$

Overall complexity $O(n)$

Searching - ordered keys

Specification for the searching problem for ordered keys:

Data $a, n, (k_i,\ i=0,n-1)$;

Precondition: $n \in N$, $n \geq 0$, and $k_0 < k_1 < \ldots < k_{n-1}$;

Results p;

Post-condition: $(p=0 \text{ and } a \leq k_0)$ or $(p=n \text{ and } a > k_{n-1})$ or $((0<p\leq n-1) \text{ and } (k_{p-1} < a \leq k_p))$.

Searching - ordered keys

```
def searchSeq(el,l):
    """
      Search for an element in a list
      el - element
      l - list of ordered elements
      return the position of first occurrence
             or the position where the element
             can be inserted
    """
    if len(l)==0:
        return 0
    poz = -1
    for i in range(0,len(l)):
        if el<=l[i]:
            poz = i
    if poz==-1:
        return len(l)
    return poz
```

$$T(n) = \sum_{(i=0)}^{(n-1)} 1 = n \in \Theta(n)$$

Searching - ordered keys


```python
def searchSucc(el,l):
    """
      Search for an element in a list
      el - element
      l - list of ordered elements
      return the position of first occurrence
             or the position where the element
             can be inserted
    """
    if len(l)==0:
        return 0
    if el<=l[0]:
        return 0
    if el>=l[len(l)-1]:
        return len(l)
    i = 0
    while i<len(l) and el>l[i]:
        i=i+1
    return i
```

Searching - ordered keys

Best case: the element is at the first position

$T(n) \in \theta(1)$

Worst-case: the element is in the n-1 position

$T(n) \in \theta(n)$

Average case: while can be executed 0,1,2,n-1 times

$T(n) = (1+2+\ldots+n-1)/n \in \Theta(n)$

Overall complexity $O(n)$

Searching algorithms

- *Sequential search*
  - Keys are successively examined
  - Keys may not be ordered
- *Binary search*
  - Uses the divide and conquer technique
  - Keys are ordered

Binary-Search (recursive)

```
def binaryS(el, l, left, right):
    """
      Search an element in a list
      el - element to be searched
      l - a list of ordered elements
      left,right the sublist in which we search
      return the position of first occurrence or
    """
    if left>=right-1:
        return right
    m = (left+right)/2
    if el<=l[m]:
        return binaryS(el, l, left, m)
    else:
        return binaryS(el, l, m, right)
```

Binary-Search (recursive)


```python
def searchBinaryRec(el, l):
    """
      Search an element in a list
      el - element to be searched
      l - a list of ordered elements
      return the position of first occurrence or
    """
    if len(l)==0:
        return 0
    if el<l[0]:
        return 0
    if el>l[len(l)-1]:
        return len(l)
    return binaryS(el, l, 0, len(l))
```

Binary-Search recurrence

$$T(n) = \begin{cases} \theta(1), & if\ n = 1 \\ T\left(\frac{n}{2}\right) + \theta(1), & otherwise \end{cases}$$

Binary-Search (iterative)


```
def searchBinaryNonRec(el, l):
    """
      Search an element in a list
      el - element to be searched
      l - a list of ordered elements
      return the position of first occurrence or the position
inserted
    """
    if len(l)==0:
        return 0
    if el<=l[0]:
        return 0
    if el>=l[len(l)-1]:
        return len(l)
    right=len(l)
    left = 0
    while right-left>1:
        m = (left+right)/2
        if el<=l[m]:
            right=m
        else:
            left=m
    return right
```

Binary-Search runtime complexity

Algorithm	running-time complexity			
	best case	worst case	average	overall
SearchSeq	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
SearchSucc	$\theta(1)$	$\theta(n)$	$\theta(n)$	$O(n)$
SearchBin	$\theta(1)$	$\theta(\log_2 n)$	$\theta(\log_2 n)$	$O(\log_2 n)$

Searching in Python


```
l = range(1,10)
try:
    poz = l.index(11)
except ValueError:
    # element is not in the list
```

- Using "in"

```
l = range(1,10)
found = 4 in l
```

Searching in Python

See **LectureXII.Search.py** for an example of overloading the **==**, **in** and creating your own collections in Python

The sorting problem

Rearrange a data collection in such a way that the elements of the collection verify a given order.

- Sort type
    - Internal sorting - the data to be sorted are available in the internal memory
    - External sorting - the data is available as a file (on external media)
    - In-place sorting - transforms the input data into the output, only using a small additional space. Its opposite is called out-of-place.
    - **Sort stability** - sorting is stable when the original order of multiple records having the same key is preserved

The sorting problem

Rearrange a data collection in such a way that the elements of the collection verify a given order.

- Elements of the data collection are called records
- A record is formed by one or more components, called fields
- A key K is associated to each record, and is usually one of the fields.
- We say that a collection of n records is:
  - Sorted in increasing order by the key K: if $K(i) \leq K(j)$ for $0 \leq i < j < n$
  - Sorted in decreasing order: if $K(i) \geq K(j)$ for $0 \leq i < j < n$

Internal sorting

Data n,K; $\{K=(k_1,k_2,\ldots,k_n)\}$

Precondition: $k_i \in R,\ i=\overline{1,n}$

Results K';

Post-condition: K' is a permutation of K, having sorted elements, i.e.

$$k'_1 \leq k'_2 \leq \ldots \leq k'_n.$$

Selection Sort

- Determine the element having the minimal key, and swapping it with the first element.
- Resume the procedure for the remaining elements, until all elements have been considered.

Selection sort algorithm

```
def selectionSort(l):
    """
        sort the element of the list
        l - list of element
        return the ordered list (l[0]<l[1]<...)
    """
    for i in range(0,len(l)-1):
        ind = i
        #find the smallest element in the rest of the list
        for j in range(i+1,len(l)):
            if (l[j]<l[ind]):
                ind =j
        if (i<ind):
            #interchange
            aux = l[i]
            l[i] = l[ind]
            l[ind] = aux
```

Selection sort - time complexity

The total number of comparisons is:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{n \cdot (n-1)}{2} \in \theta(n^2)$$

Independently of the input data, what are the best, average, worst-case computational complexities?

Selection sort - space complexity

The additional memory required (excepting the input data) is O(1).

- In-place algorithms. Algorithms that use for sorting a small (constant) quantity of extra-space (additional memory space).
- Out-of-place or not-in-space algorithms. Algorithms that use for sorting a non-constant quantity of extra-space.

Selection sort is an in-place sorting algorithm.

Direct selection sort


```python
def directSelectionSort(l):
    """
      sort the element of the list
      l - list of element
      return the ordered list (l[0]<l[1]<...)
    """
    for i in range(0,len(l)-1):
        #select the smallest element
        for j in range(i+1,len(l)):
            if l[j]<l[i]:
                swap(l,i,j)
```

Direct selection sort

Overall time complexity:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{n \cdot (n-1)}{2} \in \theta(n^2)$$

Insertion Sort

- Traversing the elements
- Insert the current element at the right position in the subsequence of already sorted elements.
- The sub-sequence containing the already processed elements is kept sorted, and, at the end of the traversal, the whole sequence will be sorted

Insertion Sort - Algorithm


```python
def insertSort(l):
    """
        sort the element of the list
        l - list of element
        return the ordered list (l[0]<l[1]<...)
    """
    for i in range(1,len(l)):
        ind = i-1
        a = l[i]
        #insert a in the right position
        while ind>=0 and a<l[ind]:
            l[ind+1] = l[ind]
            ind = ind-1
        l[ind+1] = a
```

Insertion Sort - Time complexity (WC)

Worst case - maximum number of iteration happens if the initial array is sorted in a descending order

$$T(n) = \sum_{i=2}^{n} (i - 1) = \frac{n \cdot (n - 1)}{2} \in \theta(n^2)$$

Insertion Sort - Time complexity (BC)

Best case - the initial array is already sorted

$$T(n) = \sum_{i=2}^{n} 1 = n - 1 \in \theta(n)$$

Insertion Sort - Space complexity

- Time complexity - The overall time complexity of insertion sort is $O(n^2)$.
- Space complexity - The complexity of insertion sort from the point of view of additional memory required (excepting the input data) is $\theta(1)$
- Insertion sort is an **in-place** sorting algorithm.

Bubble Sort

- Compares two consecutive elements, which, if not in the expected relationship, will be swapped.
- Comparison process will end when all pairs of consecutive elements are in the expected order relationship.

Bubble Sort - Algorithm


```python
def bubbleSort(l):
    sorted = False
    while not sorted:
        sorted = True  #assume the list is already sorted
        for i in range(1,len(l)):
            if l[i-1]>l[i]:
                swap(l, i, i-1)
                sorted = False  #the list is not sorted yet
```

Bubble Sort - Complexity

- **Best-case** running time complexity order is $\theta(n)$
- **Worst-case** running time complexity order is $\theta(n^2)$
- **Average** running-time complexity order is $\theta(n^2)$
- **Space complexity** (additional memory required, excepting the input data) is $\theta(1)$
- Bubble sort is an *in-place* sorting algorithm.

Quick Sort

Based on the *divide and conquer* technique

- **Divide:** partition array into 2 sub-arrays such that elements in the lower part $<=$ elements in the higher part.
- **Conquer:** recursively sort the 2 sub-arrays.
- **Combine:** trivial since sorting is done in place.

Partitioning: re-arrange the elements such that the element called pivot occupies the final position in the sub-sequence. If i is that position: $k_j \leq k_i \leq k_l$, for $Left \leq j < i < l \leq Right$

Quick Sort - Algorithm


```python
def partition(l,left,right):
    """
    Split the values smaller pivot greater
    return pivot position
    post:on the left we have < pivot
         on the right we have > pivot
    """
    pivot = l[left]
    i = left
    j = right
    while i!=j:
        while l[j]>=pivot and i<j:
            j = j-1
        l[i] = l[j]
        while l[i]<=pivot and i<j:
            i = i+1
        l[j] = l[i]
    l[i] = pivot
    return i
```

Quick Sort - Algorithm


```
def quickSortRec(l,left,right):

    #partition the list
    pos = partition(l, left, right)

    #order the left part
    if left<pos-1: quickSortRec(l, left, pos-1)
    #order the right part
    if pos+1<right: quickSortRec(l, pos+1, right)
```

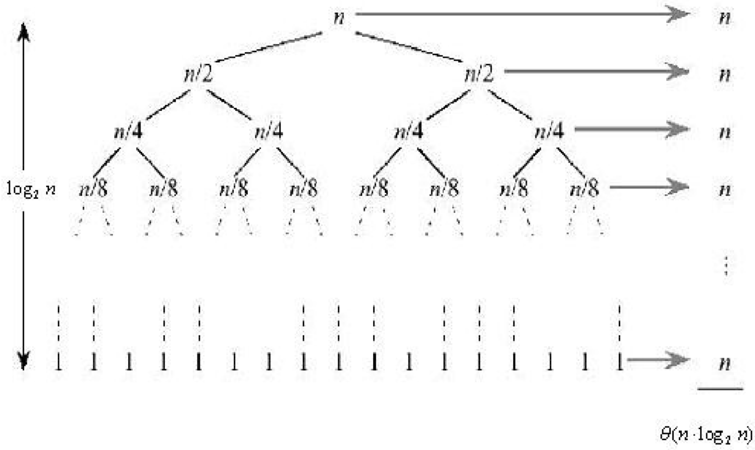
Quick Sort - Time complexity

- The running time of Quick-Sort depends on the distribution of splits
- The partitioning function Partition requires linear time
- **Best case**, the function Partition splits the array evenly:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \theta\,(n)$$

Best case complexity is: $\theta(n \log_2 n)$

Quick Sort - Best case

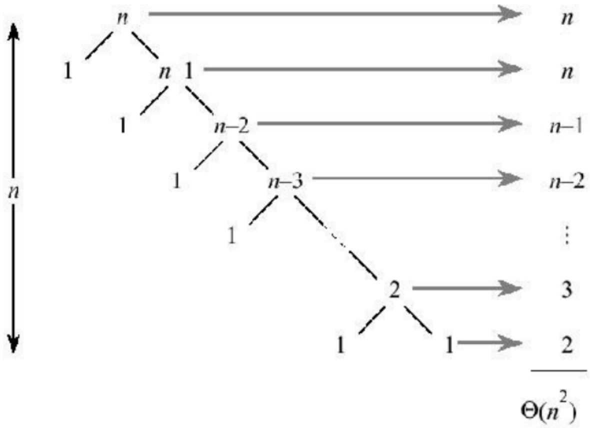


Quick Sort - Worst case

In the worst case, function Partition splits the array such that one side of the partition has only one element:

$$T(n) = T(1) + T(n-1) + \theta(n) = T(n-1) + \theta(n) = \sum_{k=1}^{n} \theta(k) \in \theta(n^2).$$

Quick Sort - Worst case



Worst case appears when the input array is sorted or is reverse sorted

Runtime complexity comparison

Lecture 12

Arthur Molnar

Reminders

Searching

The searching problem

Searching algorithms

Binary search

Searching in Python

Sorting

The sorting problem

Selection sort

Insertion sort

Bubble Sort

Quick Sort

Sorting in Python

List comprehension, lambdas

Algorithm	Complexity	
	worst-case	average
SelectionSort	$\theta(n^2)$	$\theta(n^2)$
InsertionSort	$\theta(n^2)$	$\theta(n^2)$
BubbleSort	$\theta(n^2)$	$\theta(n^2)$
QuickSort	$\theta(n^2)$	$\theta(n \cdot \log_2 n)$

Searching in Python

See **LectureXII.Sort.py** for an empirical comparison of actual sort times of different algorithms, over various data sizes

Pythonic quick Sort implementation


```python
def qsort(list):
    """
    Quicksort using list comprehensions

    """
    if list == []:
        return []
    else:
        pivot = list[0]
        lesser = qsort([x for x in list[1:] if x < pivot])
        greater = qsort([x for x in list[1:] if x >= pivot])
        return lesser + [pivot] + greater
```

List comprehension


```
[x for x in list[1:] if x < pivot]
```

```
rez = []
for x in l[1:]:
    if x<pivot:
        rez.append(x)
```

- Concise way to create lists
- Make new lists where each element is the result of some operations applied to each member of another sequence
- Consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses

Python - Optional and named arguments

- Python allows function arguments to have default values

```
def f(a=7,b = [],c="adsdsa"):
```

- If the function is called without the argument, the argument gets its default value

```
def f(a=7,b = [],c="adsdsa"):
    print a
    print b
    print c

f()
```

Console:

```
7
[]
adsdsa
```

Python - Optional and named arguments

- Arguments can be specified in any order by using named arguments.

```python
f(b=[1,2],c="abc",a=20)
```

```
Console:
20
[1, 2]
abc
```

- In Python arguments are simply a dictionary
- Need to specify a value (somehow: standard, named, default value) for each argument

Sorting in Python

L.sort([,cmp[,key[,reverse]]])

```
l.sort()
print l
```

```
l.sort(reverse=True)
print l
```

sorted(iterable[,cmp[,key[,reverse]]])

Return a new sorted list from the items in iterable.

```
l = sorted([1,7,3,2,5,4])
print l

l = sorted([1,7,3,2,5,4],reverse=True)
print l
```

```
def compare(o1,o2):
    if o1.name<o2.name:
        return -1
    if o1.name>=o2.name:
        return 1
    return 0

ls = sorted(l,compare)
```

Sorting in Python

- **cmp** - specifies a custom comparison function of two arguments (iterable elements) which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument
- **key** - specifies a function of one argument that is used to extract a comparison key from each list element
- **reverse** - is a boolean value.
- **cmp** vs. **key** - key function is called exactly once for each input record prior to making comparisons

Lambda functions in Python

With the **lambda** keyword, small anonymous functions can be created.

```
lambda x:x+7
```

Lambda forms can be used wherever function objects are required.

```
def f(x):
    return x+7

print f(5)
```

```
print (lambda x:x+7)(5)
```

Lambda functions in Python

- They are syntactically restricted to a single expression.
- Semantically, they are just syntactic sugar for a normal function definition.
- Like nested function definitions, lambda forms can reference variables from the containing scope