Lecture 10-11: Searching and Sorting

- **Searching**
- **Sorting**

Searching algorithms

- 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.

Specification for 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.

Sequential searching algorithm – keys not ordered

<table>
<tr>
<td>

```
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
```

</td>
<td>

```
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
```

</td>
</tr>
<tr>
<td>

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

</td>
<td>

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
$T(n)=(1+2+...+n-1)/n\in\Theta(n)$
Overall complexity $O(n)$

</td>
</tr>
</table>

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: $(p=0$ and $a \leq k_0)$ or $(p=n$ and $a > k_{n-1})$

or $((0<p\leq n-1)$ and $(k_{p-1} < a \leq k_p))$.

Sequential searching algorithm – ordered keys

<table>
<tr>
<td>

```python
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
```

</td>
<td>

```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
```

</td>
</tr>
<tr>
<td>

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

</td>
<td>

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
$T(n)=(1+2+...+n-1)/n\in\Theta(n)$
Overall complexity $O(n)$

</td>
</tr>
</table>

Searching algorithms

- *sequential search*
  - keys are successively examined
  - keys may not be ordered
- *binary search*
  - uses the “divide and conquer” technique
  - the keys are ordered

Binary-Search algorithm (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 the insert position
    """
    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)

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 the insert position
    """
    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

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

Binary-Search algorithm (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 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)
    right=len(l)
    left = 0
    while right-left>1:
        m = (left+right)/2
        if el<=l[m]:
            right=m
        else:
            left=m
    return right
```

Running-time 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 - index()

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

- use __eq__ or __cmp__

```
class MyClass:
    def __init__(self,id,name):
        self.id = id
        self.name = name

    def __eq__(self,ot):
        return self.id == ot.id

#     def __cmp__(self,ot):
#         return self.id.__cmp__(ot.id)


def testIndex():
    l = []
    for i in range(0,200):
        ob = MyClass(i,"ad")
        l.append(ob)

    findObj = MyClass(32,"ad")
    print "positions:" +str(l.index(findObj))
```

Searching in python- “in”

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

– need an iterable object (define __iter__ and next)

```
class MyClass2:
    def __init__(self):
        self.l = []

    def add(self,obj):
        self.l.append(obj)

    def __iter__(self):
        """
          Return an iterator object
        """
        self.iterPoz = 0
        return self
```

```
def next(self):
        """
          Return the next element in the iteration
     raise StopIteration exception if we are at the end
        """
        if (self.iterPoz>=len(self.l)):
            raise StopIteration()

        rez = self.l[self.iterPoz]
        self.iterPoz = self.iterPoz +1
        return rez

def testIn():
    container = MyClass2()
    for i in range(0,200):
        container.add(MyClass(i,"ad"))
    findObj = MyClass(20,"asdasd")
    print findObj in container
```

Performance comparison for searching

```
def measureBinary(e, l):
    sw = StopWatch()
    poz = searchBinaryRec(e, l)
    print "    BinaryRec in %f sec; poz=%i" %(sw.stop(),poz)

def measurePythonIndex(e, l):
    sw = StopWatch()
    poz = -2
    try:
        poz = l.index(e)
    except ValueError:
        pass #we ignore the error..
    print "    PythIndex in %f sec; poz=%i" %(sw.stop(),poz)

def measureSearchSeq(e, l):
    sw = StopWatch()
    poz = searchSeq(e, l)
    print "    searchSeq in %f sec; poz=%i" %(sw.stop(),poz)
```

```
search 200
    BinaryRec in 0.000000 sec; poz=200
    PythIndex in 0.000000 sec; poz=200
     PythonIn in 0.000000 sec
    BinaryNon in 0.000000 sec; poz=200
    searchSuc in 0.000000 sec; poz=200
```

```
search 10000000
    BinaryRec in 0.000000 sec; poz=10000000
    PythIndex in 0.234000 sec; poz=10000000
     PythonIn in 0.238000 sec
    BinaryNon in 0.000000 sec; poz=10000000
    searchSuc in 2.050000 sec; poz=10000000
```

Sorting

Rearrange the data collection in such a way that a certain field of the collection elements verifies a given order.

- *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)

Elements of the data collection is called *record*

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: $k_i \in R,\ i=1,n$

Results K';

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

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

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
```

Computational 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:

- best, average, worst-case computational complexity, is $\theta(n^2)$.

Space complexity

Space complexity for **selection sort**:

the additional memory required (excepting the input data) is θ (1)

- *In-place* algorithms. Algorithms that use for sorting a small (constant) quantity of extra-space (additional memory space).
- O*ut-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

```
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)
```

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 sub-sequence 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
```