Searching. Sorting. Lambda expressions.

Lect. PhD. Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

Overview

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 the **search key**.
- If the search is successful, we have the position of the record in the given sequence.
- We approach the search problem's two possibilities separately:
  - Searching with unordered keys
  - Searching with ordered keys

Searching - unordered keys

Problem specification

- **Data**: $a, n, (k_i, i = 0, .., n-1)$, where $n \in \mathbb{N}, n \geq 0$.
- **Results**: p, where $(0 \leq p \leq n-1, a = k_p)$ or $p = -1$, if key is not found.

Searching - unordered keys


```
def search_seq(el, l):
    '''
    Search for an element in list
    el - element
    l - list of elements
    Return the position of the element, -1 if not
        found
    '''
    poz = -1
    for i in range(0, len(l)):
        if el == l[i]:
            poz = i
    return poz
```

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

Searching - unordered keys

```python
def search_seq(el, l):
    '''
    Search for an element in list
    el - element
    l - list of elements
    Return the position of the element, -1 if not
        found
    '''
    i = 0
    while i<len(l) and el!=l[i]:
        i += 1
    if i<len(l):
        return i
    return -1
```

What is the difference between this and the previous version?

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: if distributing the element uniformly, the loop can be executed $0, 1, .., n-1$ times, so $T(n) = \frac{1+2+..+n-1}{n} \in \Theta(n)$.
- Overall complexity is $O(n)$

Searching - ordered keys

Problem specification

- **Data**: $a, n, (k_i, i = 0, .., n-1)$, where $n \in \mathbb{N}, n \geq 0$, and $k_0 < k_1 < ... < k_{n-1}$;
- **Results**: p, where $(p = 0$ and $a \leq k_0)$ or $(p = n$ and $a > k_{n-1})$ or $(0 < p \leq n-1)$ and $(k_{p-1} < a \leq k_p)$.

Searching - ordered keys

```
def search_seq(el, l):
    '''
    Search for an element in list
    el - element
    l - list of ordered elements
    Return the position of the first occurrence, or
        position where 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
```

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

Searching - ordered keys

```
def search_succesor(el, l):
    '''
    Search for an element in list
    el - element
    l - list of ordered elements
    Return the position of the first occurrence, or
        position where element can be inserted
    '''
    if len(l)==0 or el<=l[0]:
        return 0
    if el>=l[-1]:
        return len(l)
    i = 0
    while i<len(l) and el>l[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: if distributing the element uniformly, the loop can be executed $0, 1, .., n-1$ times, so $T(n) = \frac{1+2+..+n-1}{n} \in \Theta(n)$.
- Overabll complexity is $O(n)$

Searching algorithms

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

Recursive binary-search algorithm

```
def binary_search(key, data, left, right):
    '''
    Search for an element in an ordered list
    key - element to search
    left, right - bounds of the search
    Return insertion position of key that keeps list
        ordered
    '''
    if left >= right - 1:
        return right
    middle = (left + right) // 2
    if key < data[middle]:
        return binary_search(key, data, left, middle
            )
    else:
        return binary_search(key, data, middle,
            right)
print(binary_search(2000,data,0,len(data)))
```

Recursive binary-search function


```python
def search(key, data):
    '''
    Search for an element in an ordered list
    key - element to search
    data - the list
    Return insertion position of key that keeps list
        ordered
    '''
    if len(data) == 0 or key < data[0]:
        return 0
    if key > data[-1]:
        return len(data)
    return binary_search(key, data, 0, len(data))
```

Binary-search recurrence

- The recurrence: $\mathrm{T(n)} = \begin{cases} 1, n = 1 \\ T(\frac{n}{2}) + 1, n > 1 \end{cases}$

Iterative binary-search function

```
def binary_search(key, data):
    '''

    - specification -
    '''
    if len(data) == 0 or key < data[0]:
        return 0
    if key > data[-1]:
        return len(data)
    left = 0
    right = len(data)
    while right - left > 1:
        middle = (left + right) // 2
        if key <= data[middle]:
            right = middle
        else:
            left = middle
    return right
```

Search problem runtime complexity

Algorithm	Best case	Average	Worst case	Overall
Sequential	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Succesor	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$O(n)$
Binary-search	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	$O(\log_2 n)$

Searching in Python

Collections and search

Examine the source code in **ex29_search.py**

Iterators

Examine the source code in **ex30_iterators.py**

The sorting problem

Sorting

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

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

Demo

Stable sort example

Examine the source code in **ex31_stableSort.py**

The sorting problem

- 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

Problem specification

- **Data**: n, K, where $K = (k_1, k_2, ..., k_n), k_i \in \mathbb{R}, i = 1, n$
- **Results**: K', where K' is a permutation of K, having sorted elements: $k'_1 \leq k'_2 \leq ... \leq k'_n$.

Sorting algorithms

A few algorithms that we will study:

- Selection sort
- Insertion sort
- Bubble sort
- Quick sort

Selection Sort

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

Selection sort algorithm

```
def selection_sort(data):
    for i in range(len(data)):
        min_index = i
        # Find smallest element in the rest of the
            list
        for j in range(i+1,len(data)):
            if data[j] < data[min_index]:
                min_index = j
        data[i], data[min_index] = data[min_index],
            data[i]
```

Selection sort - time complexity

- The total number of comparisons is
$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$
- Independent of the input data size, what are the best, average, worst-case computational complexities?

Selection sort - space complexity

- **In-place** algorithms. Algorithms that use a small (constant) quantity of additional memory.
- **Out-of-place** or not-in-space algorithms. Algorithms that use a non-constant quantity of extra-space.
- The additional memory required by selection sort is $O(1)$.
- Selection sort is an in-place sorting algorithm.

Direct selection sort


```python
def direct_selection_sort(data):
    for i in range(0, len(data) - 1):
        # Select the smallest element
        for j in range(i + 1, len(data)):
            if data[j] < data[i]:
                data[i], data[j] = data[j], data[i]
```

Direct selection sort

- Overall time complexity: $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$

Insertion Sort

- Traverse 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, so that, at the end of the traversal, the whole sequence is sorted.

Insertion Sort - Algorithm


```python
def insert_sort(data):
    for i in range(1, len(data)):
        index = i - 1
        elem = data[i]
        # Insert into correct position
        while index >= 0 and elem < data[index]:
            data[index + 1] = data[index]
            index -= 1
        data[index + 1] = elem
```

Insertion Sort - time complexity

- Maximum number of iterations (worst case) happens if the initial array is sorted in a descending order:

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

Insertion Sort - time complexity

- Minimum number of iterations (best case) happens if 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 is $\theta(1)$
- Insertion sort is an **in-place** sorting algorithm.

Bubble Sort

- Compares pairs of consecutive elements that are swapped if not in the expected order.
- The comparison process ends when all pairs of consecutive elements are in the expected order.

Bubble Sort - Algorithm

```
def bubble_sort(data):
    done = False
    while not done:
        done = True
        for i in range(0, len(data) - 1):
            if data[i] > data[i+1]:
                data[i], data[i+1] = data[i+1], data
                    [i]
                done = False
```

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 is $\theta(1)$
- Bubble sort is an *in-place* sorting algorithm.

Quick Sort

Based on the *divide and conquer* technique

1. **Divide:** partition array into 2 sub-arrays such that elements in the lower part $\leq$ elements in the higher part.

Partitioning

Re-arrange the elements so 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 $Left \leq j < i < l \leq Right$

2. **Conquer:** recursively sort the 2 sub-arrays.
3. **Combine:** trivial since sorting is done in place.

Quick Sort - partitioning algorithm

```
def partition(data, left, right):
    pivot = data[left]
    i = left
    j = right
    while i != j:
        # Find an element smaller than the pivot
        while data[j] >= pivot and i < j:
            j -= 1
        data[i] = data[j]
        # Find an element larger than the pivot
        while data[i] <= pivot and i < j:
            i += 1
        data[j] = data[i]
    # Place the pivot in position
    data[i] = pivot
    return i
```

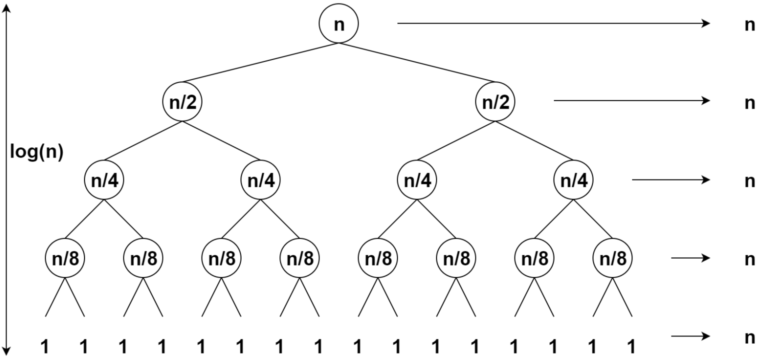
Quick Sort - algorithm


```
def quick_sort(data, left, right):
    # Partition the list
    pos = partition(data, left, right)
    # Order left side
    if left < pos - 1:
        quick_sort(data, left, pos - 1)
    # Order right side
    if pos + 1 < right:
        quick_sort(data, pos + 1, right)
```

Quick Sort - time complexity

- The run time of quick-sort depends on the distribution of splits
- The partitioning function requires linear time
- **Best case**, the partitioning function splits the array evenly: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$, $T(n) \in \Theta(n \log_2 n)$

Quick Sort - best partitioning

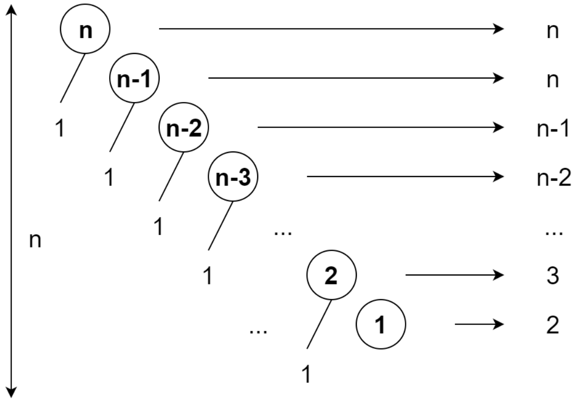


- We partition n elements $\log_2 n$ times, so $T(n) \in \Theta(n \log_2 n)$

Quick Sort - worst partitioning

- 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 partitioning appears when the input array is sorted or reverse sorted, so n elements are partitioned n times, $T(n) \in \Theta(n^2)$

Sorting runtime complexity

Algorithm	Worst case	Average
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Bubble sort	$\Theta(n^2)$	$\Theta(n^2)$
Quick sort	$\Theta(n^2)$	$\Theta(n \log_2 n)$

Demo

Sorting

Examine the source code in **ex32_sort.py**

Lambda expressions

Lambda expressions

Small anonymous functions, that you define and use in the same place.

- Syntactically restricted to a single expression.
- Can reference variables from the containing scope (just like nested functions).
- They are *syntactic sugar* for a function definition.

Demo

Lambda Expressions

Examine the source code in **ex33_lambdas.py**