Sorting

- **Sorting algorithms and complexity**
- **Sorting in Python**

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

```
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

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

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

Average case: $\frac{n^2 + 3 \cdot n}{4} - \sum_{i=1}^{n} \frac{1}{i} \in \theta(n^2)$

for a fixed *i* and any *k*, $1 \le k \le i$, the probability that x_i is the *k*-th greater element from the sub-sequence $x_1, x_2, \ldots, x_i$ is $\frac{1}{i}$

So, for a fixed i, we can deduce the following:

The number of **while** iterations	The probability that the number of **while** iterations is the one from the first column	Possible cases
1	$\frac{1}{i}$	there is a single case that **while** is repeated once: $x_i < x_{i-1}$
2	$\frac{1}{i}$	there is a single case that **while** is repeated twice: $x_i < x_{i-2}$
...	$\frac{1}{i}$	**...**
i-1	$\frac{2}{i}$	there are two cases that **while** is repeated **i-1** times: $x_i < x_1$ and $x_1 \le x_i < x_2$

It follows that the average number of comparisons (iterations of **while**) for a fixed i is calculated as:

$$c_i = 1 \cdot \frac{1}{i} + 2 \cdot \frac{1}{i} + \ldots + (i-2) \cdot \frac{1}{i} + (i-1) \cdot \frac{2}{i} = \frac{i+1}{2} - \frac{1}{i}$$

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

the initial array is already sorted

Insertion Sort

- 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

```
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 $\theta(n^2)$.

Overall time complexity of *BubbleSort* is $O(n^2)$

Space complexity (additional memory required, excepting the input data) is $\theta(1)$.

- *bubble sort* is an ***in-place*** sorting algorithm.

QuickSort

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:

$$k_j \leq k_i \leq k_l \text{ , for } Left \leq j < i < l \leq Right$$

Quick-Sort algorithm

```
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
```

```
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)
```

QuickSort – Time complexity

The running time of *Quick-Sort* depends on the distribution of splits

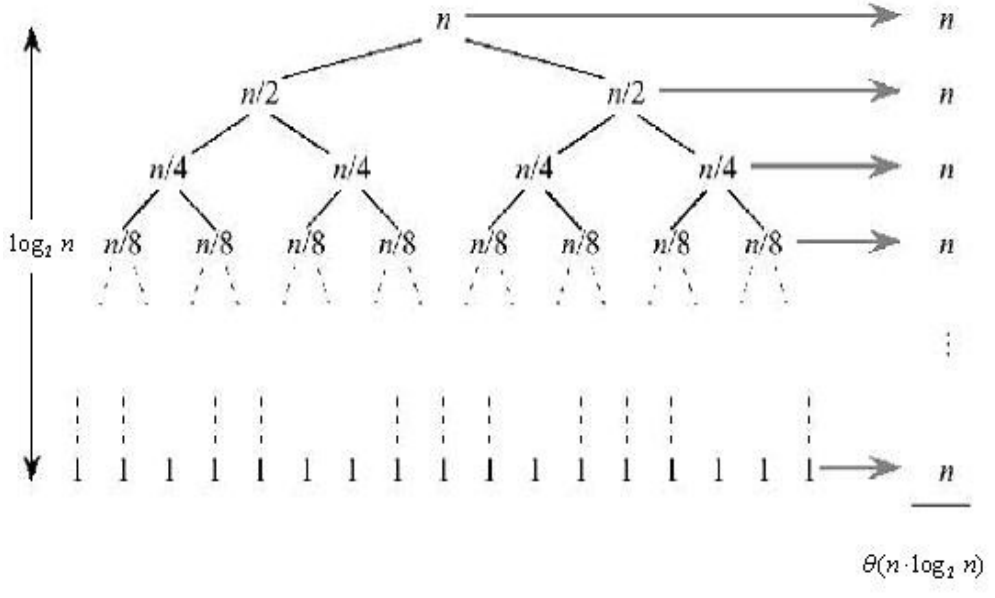
The partitioning function Partition requires linear time

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 $\theta(n \cdot \log_2 n)$.

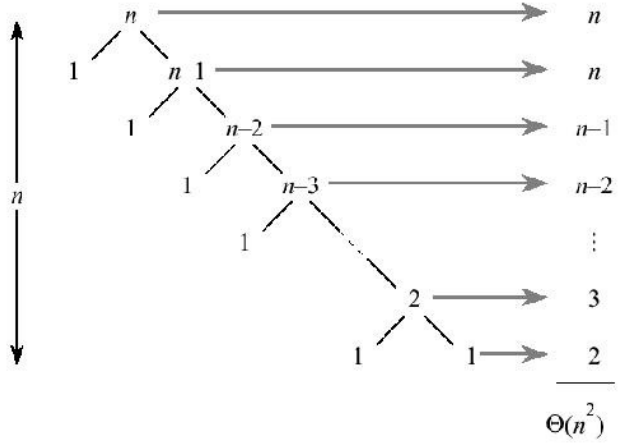
QuickSort – Best case



QuickSort – Worst case

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

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



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

QuickSort - Average case

Suppose that alternate the preceding two cases:

- the favorable (lucky) case with $\theta(n \cdot \log_2 n)$ time complexity (we denote it by L)
- and the unfavorable (unlucky) case with $\theta(n^2)$ time complexity (we denote it by U).

In other words, we have the following recurrence:

$$\begin{cases} L(n) = 2 \cdot U\left(\frac{n}{2}\right) + \theta(n) & \textit{lucky case} \\ U(n) = L(n-1) + \theta(n) & \textit{unlucky case} \end{cases}$$

We consequently get that

$$L(n) = 2 \cdot \left(L\left(\frac{n}{2} - 1\right) + \theta\left(\frac{n}{2}\right) \right) + \theta(n) = 2 \cdot L\left(\frac{n}{2} - 1\right) + \theta(n) = \theta(n \cdot \log_2 n),$$

meaning that the average time complexity is $T(n) = L(n) \in \theta(n \cdot \log_2 n)$.

Python - Quick-Sort

```
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 comprehensions

```
[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

Running-time complexity of the analyzed sorting algorithms:

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	$\theta(n^2)$	$\theta(n \cdot \log_2 n)$

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
```

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

```
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 - list.sort() / build in function: sorted

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)
```

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

Sort stability

stable **-** when multiple records have the same key,their original order is preserved

Python - lambda functions (anonymous functions, lambda form)

- 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)
```

- 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:

```
    l = []
    l.append(MyClass(2,"a"))
    l.append(MyClass(7,"d"))
    l.append(MyClass(1,"c"))
    l.append(MyClass(6,"b"))
    #sort on name
    ls = sorted(l,key=lambda o:o.name)
    for x in ls:
        print x
```

```
    l = []
    l.append(MyClass(2,"a"))
    l.append(MyClass(7,"d"))
    l.append(MyClass(1,"c"))
    l.append(MyClass(6,"b"))
    #sort on id
    ls = sorted(l,key=lambda o:o.id)
    for x in ls:
        print x
```

TreeSort

The algorithm essentially builds a binary tree with the property that at any node of the tree, its left sub-tree has articles with keys less than the node key, and right sub-tree has articles with keys greater than the node key.

With an in order traversal of this tree, we will have our elements traversed in increasing order of their keys.

The tree will be built by successive insertions of elements in the left or right of the current node, according to the relation between the root key of the current sub-tree and the key of the inserted element.

We remark that any insertion takes place at the terminal nodes of the tree.

MergeSort

Is based, as well, on the “divide and conquer” approach.

The sequence to be sorted is split in two sub-sequences, to be sorted separately. The sorted sub-sequences will then be merged into the final sorted sequence.

Each sub-sequence will be sorted using the same approach until we get such sub-sequences that can elementarily be sorted without splitting them (i.e. they have only one element).

Merging

Data m, (x_i, i=1,m), n, (y_i, i=1,n);

Precondition: {$x_1 <= x_2 ... <= x_m$} and {$y_1 <= y_2 <= ... <= y_n$}

Results k, (z_i, i=1,k);

Post-condition: {$k=m+n$} and {$z_1 <= z_2 <= ... <= z_k$} and ($z_1, z_2, ..., z_k$) is a permutation of the values ($x_1, ..., x_m, y_1, ..., y_n$)

Running time complexity of the Merge: $\theta(m+n)$.

The extra space complexity required by the Merge is $\theta(1)$