University of Applied Sciences and Arts of Southern Switzerland
Department of Innovative Technologies

SUPSI

Algorithm analysis

How can we decide which algorithm is better?

- Measure execution time?
  - Store the value of the computer internal clock
  - Launch the algorithm and wait for its completion
  - Read the new value of the clock, compute the difference

How can we decide which algorithm is better?

- Measure execution time?
  - Store the value of the computer internal clock
  - Launch the algorithm and wait for its completion
  - Read the new value of the clock, compute the difference

- Problem: it depends on external factors
  - on the amount of data to be processed
  - on the other tasks the computer is running at the same time
  - on the hardware
  - on the programming language and compiler

Complexity analysis

Complexity analysis

- Instead of an empirical approach, base the analysis on the **structure** of the algorithm
- Measure the aspects that most critically affect its execution time
  - Number of logical comparisons
  - Number of data assignments
  - Number of arithmetic operations

- Example: the following structure contains $2n^2$ additions (n*2n)

```
totalSum = 0 # Version 1
for i in range(n) :
    rowSum[i] = 0
    for j in range( n ) :
        rowSum[i] = rowSum[i] + matrix[i,j]
        totalSum = totalSum + matrix[i,j]
```

n iterations

2n additions

Can we improve the algorithm?

- Moving one line out of the loop

```
totalSum = 0 # Version 2
for i in range( n ) :
    rowSum[i] = 0
    for j in range(n) :
        rowSum[i] = rowSum[i] + matrix[i,j]
    totalSum = totalSum + rowSum[i]
```

n iterations

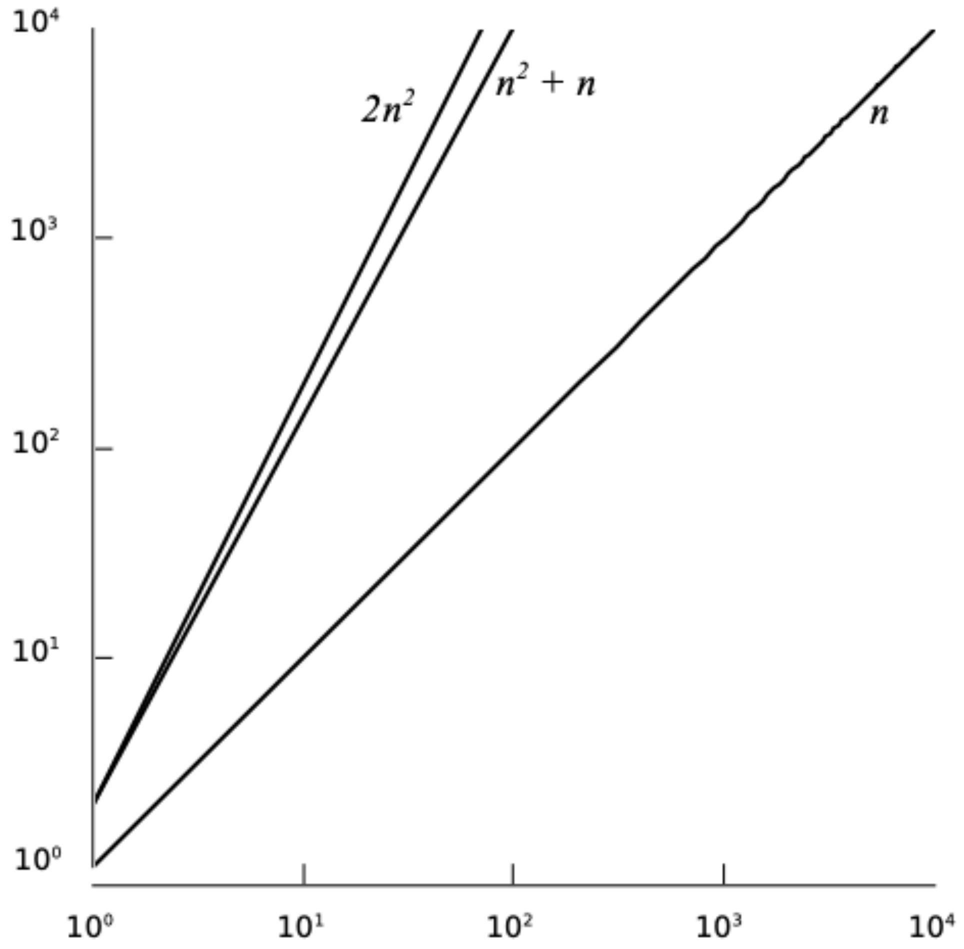
n additions

1 addition

- We have a total of n*(n+1) additions
- For n large this is a <u>significative improvement</u>

A comparison of the two algorithms

n	$2n^2$	$n^2 + n$
10	200	110
100	20'000	10'100
1000	2'000'000	1'001'000
10000	200'000'000	100'010'000
100000	20'000'000'000	10'000'100'000



log-log scale

Big-O notation

- Computer scientists use the big-O notation which considers the **order of magnitude** rather than the number of individual operations

- Assume we have a function that expresses the number of steps required by an algorithm.

- Let's suppose that exists a function $f(n)$ defined for n>=0 such that, for some constant c and m, $T(n) \leq cf(n)$, for all sufficiently large values of $n \geq m$

- In such a case the algorithm is said to have a **time-complexity** of (or to execute in the order of) **f(n)**

- In other words, there is a positive integer m and a constant c (constant of proportionality) such that for all n ≥ m, $T(n) \leq cf(n)$.

Big-O notation

- $O(f(n))$ is the notation used to specify that an algorithms has a time complexity (or runs in the order) of f(n)

- Example 1
  - $T_1(n) = 2n^2$
  - With c=2 we have: $2n^2 \leq 2n^2$ T(n) c f(n)
  - and therefore we get $O(n^2)$

- Example 2
  - $T_2(n) = n^2 + n$
  - With c=2 we have $n^2 + n \leq 2n^2$ T(n) c f(n)
  - and therefore we get again $O(n^2)$

- We chose c=2 as it was the smallest integer that satisfied the equation $T(n) \leq cf(n)$ in both cases

Big-O notation

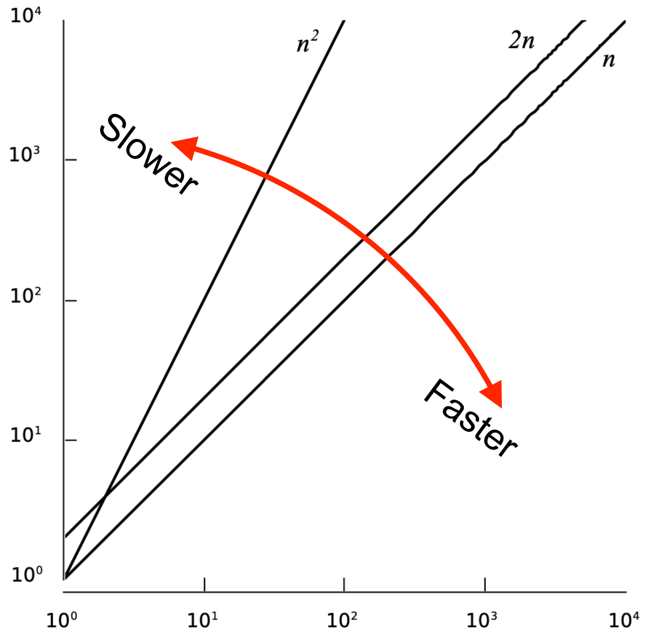
- The function $f(n) = n^2$ is not the only choice
- We could have chosen $f(n) = n^3$ or $f(n) = n^4 + \sqrt{n}$ or any other majorizing function
- $f(n) = n^2$ provides the the **tightest (lowest) upper bound** or limit for the run time of an algorithm
- The big-O notation indicates the algorithm's efficiency for large values of n

Constant of proportionality

- The constant c is crucial only if two algorithms have the same f(n)

- Example:
  - Algorithm L1 has growth rate n^2, the time complexity is $O(n^2)$ with c=1
  - Algorithm L2 has growth rate $2n$, the time complexity is $O(n)$ with c=2
  - L1 is slower even if c is smaller!

n	n^2	$2n$
10	100	20
100	10,000	200
1000	1,000,000	2,000
10000	100,000,000	20,000
100000	10,000,000,000	200,000



Constructing T(n)

- Let's evaluate an algorithm by evaluating every single operation performed
- Assume that **each atomic operation executes in constant time**
- The total number of time required to execute an algorithm is:
  - $T(n) = f_1(n) + f_2(n) + \ldots + f_k(n)$

(a)

```
1          totalSum = 0
n      1   for i in range( n ) :
       1     rowSum[i] = 0
    n  1     for j in range( n ) :
       1       rowSum[i] = rowSum[i] + matrix[i,j]
       1       totalSum = totalSum + matrix[i,j]
```

(b)

```
n      for i in range( n ) :
          ...
   n   for j in range( n ) :
          ...
```

Choosing the function f(n)

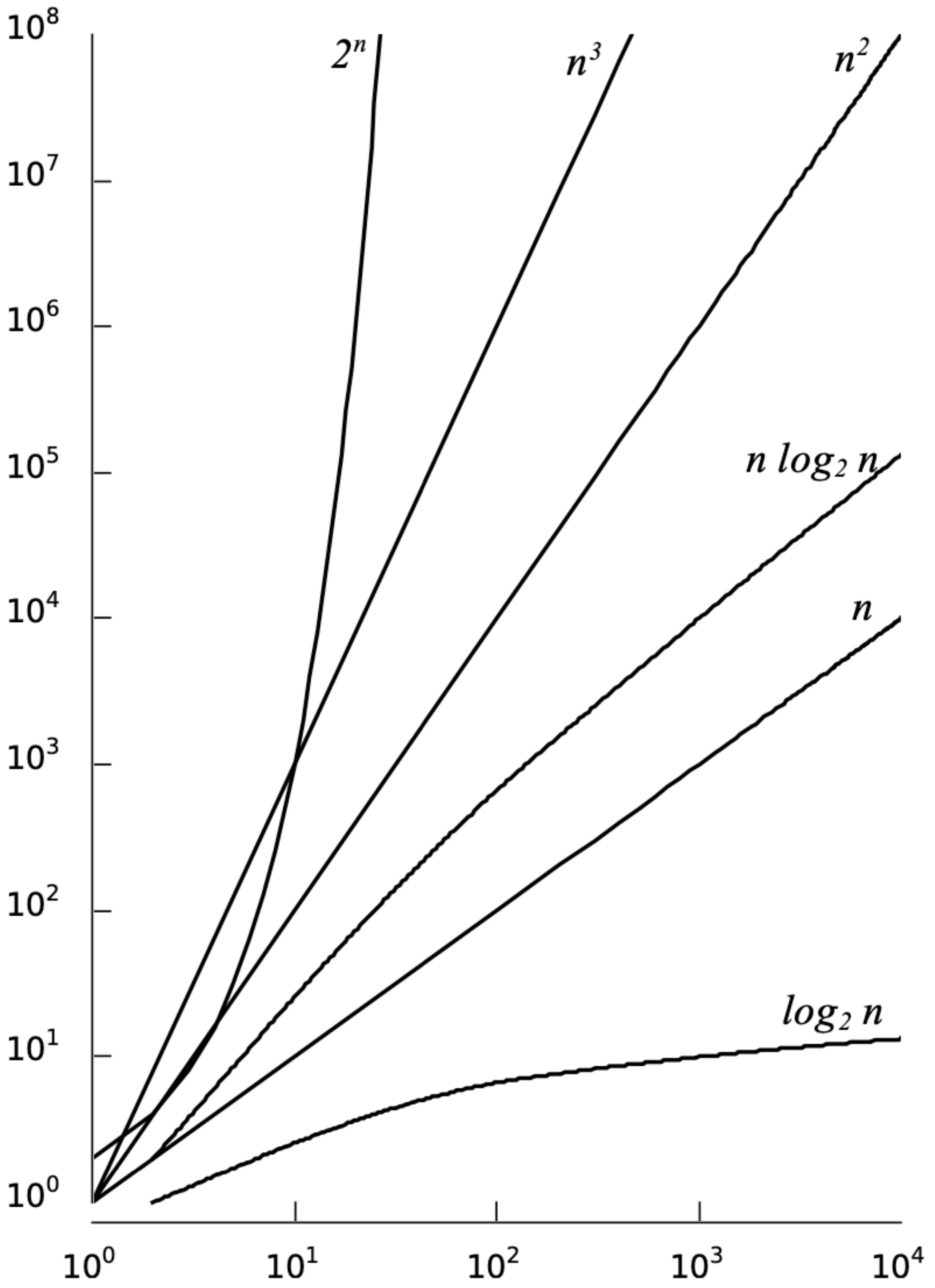
- The function f(n) to be used to compute the complexity is the **dominant term** within T(n)

- For instance:
  - If $T(n) = n^2 + \log_2 n + 3n$
  - The dominant term is n^2 because for $n \geq 3$ then
    - $n^2 + \log_2 n + 3n < n^2 + n^2 + n^2$
    - $n^2 + \log_2 n + 3n < 3n^2$
  - And the time complexity is $O(n^2)$

- Why dominant? Assume $T(n) = n^2 + 15n + 500$. if n is small, then 500 is the most important term, but as n increases n^2 will become the most relevant term

Classes of algorithms

$f(\cdot)$	Common name
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Log-linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential



Classes of algorithms

- The various classes are named on the basis of the big-O time complexity

- A **logarithmic** algorithm is an algorithm whose big-O time complexity is $O(\log_a n)$
  - In many computer science algorithms a=2
  - Logarithmic algorithms are very efficient as they grow less than n

- **Polynomial** algorithms have a time complexity expressed as:
  - $a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$
  - Their time complexity is expressed as $O(n^m)$
    - Most common cases are **linear**, **quadratic** and **cubic**

- Finally **exponential** algorithms with a time complexity $O(a^n)$ are the worst in terms of performance

Evaluation of Python code

- Basic operations require constant time
- A basic operation is a statement or a function call whose execution time does not depend on the specific value of the data
- `x=5` is a basic operation as it requires the same number of instruction on the CPU to be executed, irrespectively of the assigned value
- ```
  y=x
  z=x+y*6
  done = x > 0 and x < 100
  ```
  are all basic operations, as well as the subscript operator
  ```

Linear time examples

- The assignment instruction `y=ex1(n)` requires constant time, but this does not include the time spent within the function call `ex1(n)`
- In the function ex1:
  - The loop has $T(n) = n \cdot 1$
  - And we have complexity O(n)
  - The assignment and the return have constant time
- We focus on loops and repetitions
- In the function ex2:
  - There are two loops $T(n) = n + n$
  - But the complexity is still O(n)

```
def ex1( n ):
    total = 0
    for i in range( n ) :
        total += i
    return total
```

```
def ex2( n ):
    count = 0
    for i in range( n ) :
        count += 1
    for j in range( n ) :
        count += 1
    return count
```

Quadratic time examples

- The time required by the inner loop is $T_i(n) = n$
- The time required by the outer loop is $T_o(n) = n \cdot T_i(n) = n \cdot n = n^2$
- The time complexity is $O(n^2)$

- Not all nested loops are quadratic
  - The inner loop executes a constant number of times, it executes in **constant time**
  - The time complexity is O(n)

```python
def ex3(n):
    count = 0
    for i in range(n) :
        for j in range(n) :
            count += 1
    return count
```

```python
def ex4(n):
    count = 0
    for i in range(n):
        for j in range(25):
            count += 1
    return count
```

Quadratic time examples: a special case

- How many times do we execute the loop?
- Example: n=3
    - i=1; j=1,2
    - i=2; j=1,2,3
    - i=3; j=1,2,3,4
- In general $T(n) = n \cdot (n+1) = n^2 + n$
- So the complexity is $O(n^2)$

```python
def ex5(n):
    count = 0
    for i in range(n):
        for j in range(i+1):
            count += 1
    return count
```

Logarithmic time examples

- In ex6 the loop variable *i* is cut in half at each iteration
- It follows an "exponential decay"
- The number of iterations required to get i==1 is $\lfloor \log_2 n \rfloor + 1$ (largest integer smaller than log2 +1)
- The function has therefore $O(\log n)$ complexity

```
def ex6(n):
    count = 0
    i = n
    while i >= 1:
        count += 1
        i = i // 2
    return count
```

- In ex7 the inner loop is executed n times, but it calls the function ex6 which has a complexity log n, so we get $O(n \log n)$

```
def ex7(n):
    count = 0
    for i in range(n)
        count += ex6(n)
    return count
```

Different cases

- Some algorithms can have run times that are different orders of magnitude for different sets of inputs of the same size.
- These algorithms can be evaluated for their best, worst, and average cases.
- These algorithms usually have an event controlled loop or a switch statement

- Example:
  - If a list does not contain any negative number the algorithm scans all values (worst case)
  - If a list has a negative number in the first position it runs in constant time (best case)
  - The average case depends on the data we typically feed to the algorithm

```
def findNeg( intList ):
    n = len(intList)
    for i in range(n):
        if intList[i] < 0:
            return i
    return None
```

Evaluating the Python list

Analysing the performance of ADTs: the Python list

- ADTs are widely re-used
- Optimising their code and improving their efficiency has a great impact
- We start by analysing the Python List

List operation	Worst case
`v = list()`	O(1)
`v = [0]*n`	O(n)
`v[i] = x`	O(1)
`v.append(x)`	O(n)
`v.extend(w)`	O(n)
`v.insert(x)`	O(n)
`v.pop()`	O(n)
traversal	O(n)

List performance: traversal

```python
sum = 0
for elem in myList:
    sum = sum+elem
```

An example of list traversal

```python
sum = 0
for i in range(len(myList)):
    sum = sum + myList[i]
```

A possible implementation

- It is obvious that the complete list traversal has a time complexity of $O(n)$

List performance: allocation

- List allocation means creating memory space to store all list elements
- There are two possibilities
    - Creation of an empty list:

```
temp = list()
```

    - This happens in constant time
    - Creation of a list on n elements, each one initialised to 0

```
valueList = [ 0 ] * n
```

    - The allocation is in constant time but the traversal takes linear time

List performance: appending, extending, inserting

- Appending to a list
  - If there is space in the list, constant time
  - If the list is full, the space must be allocated and while the allocation is O(1), copying the old values in the new array takes O(n)

- Extending a list
  - The extend() operation adds the entire contents of a source list to the end of the destination list.
  - If the destination list has enough space, the cost is T(n)=n, but if new space must be allocated T(n)=n+n (worst case), but in both cases we have O(n)

- Inserting in a list
  - Inserting a new element can require shifting elements, which requires linear time, so O(n)

Amortised cost

What does it mean amortised cost?

- If we append to a list and there is space, this happens in **constant time**.
- If there is no space, we must allocate it, this happens in **linear time**.
- How much space do we allocate? Just what is needed? Or a bit more?
- We can devise a strategy to expand the list in order to minimise future reallocations, thus reducing the cost.
- Let's consider the following code:

```
n=16
L = list()
for i in range(1, n+1):
    L.append(i)
```

- Let's suppose that each time the space is finished, the array is doubled in size
- Let's compute the time for each individual operation in the loop (**aggregate method**)

The aggregate method

- The array doubles the size when $i = 2^k + 1$ for $k \in N$
- s_i is the time required to store an item
- e_i is the time required to expand the array
- The total storage cost s_i is 16, the expansion cost e_i is 15
- For any n $s_i + e_i < 2n$
- Since there are a few append operations we distribute them across the n elements so the amortised cost is $T(n) = 2n/n$
- The amortised time complexity is O(1)

i	s_i	e_i	**Size**	**List Contents**
1	1	-	1	1
2	1	1	2	1 2
3	1	2	4	1 2 3
4	1	-	4	1 2 3 4
5	1	4	8	1 2 3 4 5
6	1	-	8	1 2 3 4 5 6
7	1	-	8	1 2 3 4 5 6 7
8	1	-	8	1 2 3 4 5 6 7 8
9	1	8	16	1 2 3 4 5 6 7 8 9
10	1	-	16	1 2 3 4 5 6 7 8 9 10
11	1	-	16	1 2 3 4 5 6 7 8 9 10 11
12	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12
13	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13
14	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14
15	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

The aggregate method

- The array doubles the size when $i = 2^k + 1$
- s_i is the time requi
- e_i is the time requ
- The storage cost i is 15
- For any n $s_i + e_i$
- Since there are a few aspen operations we distribute them across the n elements so the amortised cost is $T(n) = 2n/n$
- The time complexity is O(1)

Warning! Amortized cost is not Average Case Time

In average case analysis, the evaluation is done by computing an average over all possible inputs and sometimes requires the use of statistics. Amortized analysis computes an average cost over a sequence of operations in which many of those operations are "cheap" and relatively few are "expensive" in terms of contributing to the overall time.

i	s_i	e_i	Size	List Contents
1	1	-	1	1
2	1	1	2	1 2
3	1	2	4	1 2 3
4	1	-	4	1 2 3 4
5	1	4	8	1 2 3 4 5
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
10	1	-	16	1 2 3 4 5 6 7 8 9 10
11	1	-	16	1 2 3 4 5 6 7 8 9 10 11
12	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12
13	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13
14	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14
15	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Evaluating the Set ADT

Analysing the performance of ADTs: the Set ADT

- Simple operations
  - Creation and length can be performed in constant time
  - `__contains__` (used by `in`) performs a linear search (worst case O(n))
  - add is O(n) since the element cannot be a duplicate and it must be searched for
- Operations on two sets
  - These methods all have two nested for, their time complexity will be $O(n^2)$
- Set union: $O(n^2)$
  - it requires three steps:
  1. Create the new set (constant time)
  2. Fill the new set with the element from the first set (linear time)
  3. Iterate over the elements of the second set to check whether they are in the first set

Set operation	Worst case
`s = Set()`	O(1)
`len(s)`	O(1)
`x in s`	O(n)
`s.add(x)`	O(n)
`s.isSubsetOf(t)`	$O(n^2)$
`s==t`	$O(n^2)$
`s.union(t)`	$O(n^2)$
traversal	O(n)

Application: the sparse matrix

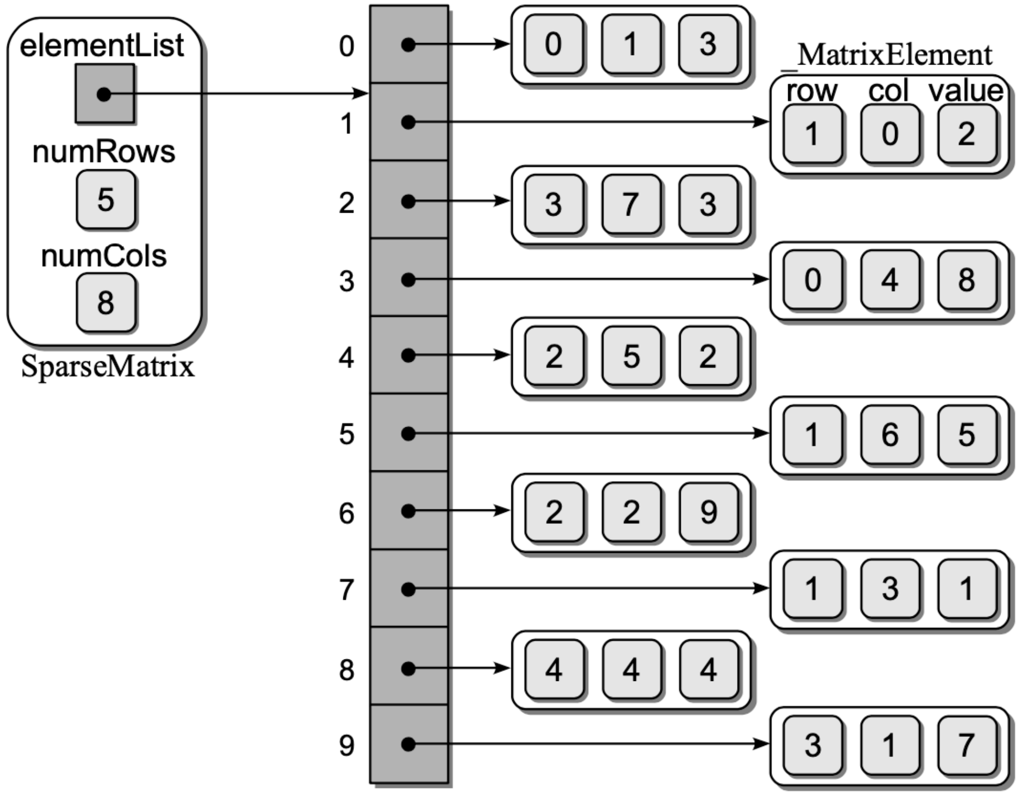
What is a sparse matrix?

- A sparse matrix is a matrix with k values with k << m*n (m number of rows, n number of columns)
- The Matrix ADT works well for general, non-sparse, matrices
- A sparse matrix may waste lots of memory space and operations are also inefficient (lots of zeros still being used in multiplications etc.)
- Can we devise a data structure which is more efficient for storing sparse matrices?

$$\begin{bmatrix} \cdot & 3 & \cdot & \cdot & 8 & \cdot & \cdot & \cdot \\ 2 & \cdot & \cdot & 1 & \cdot & \cdot & 5 & \cdot \\ \cdot & \cdot & 9 & \cdot & \cdot & 2 & \cdot & \cdot \\ \cdot & 7 & \cdot & \cdot & \cdot & \cdot & \cdot & 3 \\ \cdot & \cdot & \cdot & \cdot & 4 & \cdot & \cdot & \cdot \end{bmatrix}$$

A list-based implementation

(i,j)	0	1	2	3	4	5	6	7
0	0	**3**	0	0	**8**	0	0	0
1	**2**	0	0	**1**	0	0	**5**	0
2	0	0	**9**	0	0	**2**	0	0
3	0	**7**	0	0	0	0	0	**3**
4	0	0	0	0	**4**	0	0	0



A list-based implementation of the SparseMatrix ADT: constructor

```python
# Implementation of the Sparse Matrix
class SparseMatrix:
    # Create a sparse matrix of
    def __init__(self, numRows, numCols):
        self._numRows = numRows
        self._numCols = numCols
        self._elementList = list()

    # Return the number of rows
    def numRows(self):
        return self._numRows

    # Return the number of columns in
    def numCols(self):
        return self._numCols
```

```python
# Storage class for holding the non-zero
matrix elements.
class _MatrixElement:
    def __init__(self, row, col, value):
        self.row = row
        self.col = col
        self.value = value
```

This is a list of MatrixElement

SparseMatrix ADT: setter

```
# Set the value of element (i,j) to the value s: x[i,j] = s
def __setitem__(self, ndxTuple, scalar):
    ndx = self._findPosition(ndxTuple[0], ndxTuple[1])
    if ndx is not None:  # if the element is found in the list.
        if scalar != 0.0:
            self._elementList[ndx].value=scalar
        else:
            self._elementList.pop(ndx)  # remove the index
    else:  # if the index is not already in the list.
        if scalar != 0.0:
            element = _MatrixElement(ndxTuple[0], ndxTuple[1], scalar)
            self._elementList.append(element)
```

SparseMatrix ADT: the findPosition helper method

```python
# Helper method used to find a specific matrix element (row,col) in the
# list of non-zero entries. None is returned if the element is not found.
def _findPosition(self, row, col):
    n = len(self._elementList)
    for i in range(n):
        if row == self._elementList[i].row and \
                col == self._elementList[i].col:
            return i  # return the index of the element if found.
    return None  # return None when the element is zero.
```

SparseMatrix ADT: getter

```python
# Return the value of element (i,j) of the matrix: x[i,j]
def __getitem__(self, ndxTuple):
    ndx = self._findPosition(ndxTuple[0], ndxTuple[1])
    if ndx is not None:
        return self._elementList[ndx].value
    else:
        return 0
```

SparseMatrix ADT: addition

- The standard matrix summation iterates over two loops
- If the matrix is sparse, this $O(n^2)$ algorithm is inefficient

```
def __add__( self, rhsMatrix ):
    assert self.numRows()==rhsMatrix.numRows() and \
           self.numCols()==rhsMatrix.numCols(), \
            "matrix sizes not compatible"
    newMatrix = SparseMatrix(self.numRows(), self.numCols())
    for r in range(self.numRows()):
        for c in range(self.numCols()):
            newMatrix[r, c] = self[r, c] + rhsMatrix[r, c]
    return newMatrix
```

SparseMatrix ADT: an improved version for matrix addition

- Verify the size of the two matrices to ensure they are the same as required by matrix addition.
- Create a new SparseMatrix object with the same number of rows and columns as the other two.
- Duplicate the elements of the self matrix and store them in the new matrix.
- Iterate over the element list of the righthand side matrix (rhsMatrix) to add the non-zero values to the corresponding elements in the new matrix.

SparseMatrix ADT: efficiency analysis

- We assume a square n x n matrix
- `_findPosition()_`: it performs a linear search over $k << n^2$ elements. The worst case run is O(k)
- `_setitem()_` and `_getitem()_` use `_findPosition()_` and therefore they are both O(k)
- _add()_
  - Size verification and new matrix creation happen in constant time
  - Duplicating the entries of LHS requires k time as append has an amortised cost of O(1)
  - The second loop requires getting and setting the value (2k) and this is repeated for k elements: $2k^2$ time
  - $O(k^2)$ is the worst case

- For the sum of two matrices if $k= n^2$ we would get a time complexity of $O(n^4)$

A comparison of the time complexity of Matrix against SparseMatrix

Operation	Matrix	SparseMatrix
constructor	O(1)	O(1)
`s.numRows()`	O(1)	O(1)
`s.numCols()`	O(1)	O(1)
`s.scaleBy()`	$O(n^2)$	O(k)
`x=s[i,j]`	O(1)	O(k)
`s[i,j]=x`	O(1)	O(k)
`r=s+t`	$O(n^2)$	$O(k^2)$

Hands-on activity (20 min)

- Take a look at the code of the Sparse Matrix class presented in the textbook and implement it in two Python files
    - testmatrix.py - containing the main body that will test the Sparse Matrix
    - sparsematrix.py which contains the ADT interface and implementation
- Is important that you read the code that has been provided and you understand its content.
- BEWARE: there may be a few mistakes in the code of the book, fix them!