SEMINAR XI – RECURSION. ALGORITHM COMPLEXITY

WHAT YOU SHOUD KNOW AFTER ATTENDING

- Basics of recursion. Direct and indirect recursion
- The importance of determining algorithm complexity.
- The link between mathematically determined complexity and results observed empirically
- Determining the time complexity for certain algorithms

RECURSION. EXAMPLES

- Steps to look for in a successful recursive implementation:
  1. One basic, simple case
  2. Reduction works towards the simple case
- Direct vs. Indirect recursion
  - ***Direct*** – Function calls itself
  - ***Indirect*** – Function calls another function, but it will be called again

SUM OF FIRST N NUMBERS

```
def sum(n):
    if n == 0:
        return 0
    return n + sum(n-1)
```

EVEN VS. ODD NUMBERS

- An easy example for indirect recursion

```
def isOdd(n):
    if n == 0:
        return False
    return isEven(n-1)

def isEven(n):
    if n == 0:
        return True
      return isOdd(n-1)
```

FIBBONACI SEQUENCE (ITERATIVE, RECURSIVE + RUNTIME ANALYSIS)

```
def fib_rec(n):
    if n == 0:
        return 0
    if n==1:
        return 1
```

```
        return fib_rec(n-1) + fib_rec(n-2)

    def fib_iter(n):
        sum1 = 1
        sum2 = 1
        rez = 0
        for i in range(2, n+1):
            rez = sum1+sum2
            sum1 = sum2
            sum2 = rez
        return rez
```

WHICH FIB(N) IMPLEMENTATION IS FASTER? WHY?

- Examine what happens for the ***fib(4)*** recursive calculation, for both the iterative as well as recursive implementations. Run the code below (with the ***fib_iter*** and ***fib_rec*** functions defined above) and examine the runtimes.

```
def play_fib():
    terms = [10, 20, 25, 30, 35]
    for t in terms:
        t1 = time()
        fib_iter(t)
        t2=time()
        fib_rec(t)
        t3=time()
        print "Iter " + str(t) + " terms in = "+str(t2-t1)
        print "Rec " + str(t) + " terms in = "+str(t3-t2)
```

N	Iterative	Recursive
10	0.0	0.0
20	0.0	0.01
25	0.0	0.13
30	0.0	1.43
35	0.0	15.97

DETERMINING ALGORITHM COMPLEXITY

Determine what is the time complexity for the algorithms below.

SIMPLE CASES

```
found ← false
for i ← 1, n do
    if x_i = a then
        found ← true
    endif
endfor
```

```
found ← false
while found = false do
    if x_i = a then
        found ← true
    endif
endwhile
```

NESTED LOOPS

$s \leftarrow 0$
for $i \leftarrow 1, n^2$ do
    $j \leftarrow i$
    while $j \neq 0$ do
        $s \leftarrow s + j$
        $j \leftarrow j - 1$
    endwhile
endfor

Answer
For a given *i*, the *while* body runs *i* times. Therefore, the total time is given using the expression:

$$\sum_{i=1}^{n^2} i = \frac{n^2(n^2+1)}{2} \Rightarrow complexity\ \boldsymbol{\theta(n^4)}$$

SOME PYTHON CODE

```python
def f2(l):
    sum = 0
    for el in l:
        j = len(l)
        while j>1:
          sum+=el*j
          j=j//3
    return sum
```

```python
def sum_rec(l):
    if l==[]:
        return 0
    if len(l)==1:
        return l[0]
    m = len(l)//2
    return sum_rec(l[:m])+ sum_rec(l[m:])
```

APPROXIMATION

$s \leftarrow 0$
for $i \leftarrow 1, n^2$ do
    $j \leftarrow i$
    while $j \neq 0$ do
        $s \leftarrow s + j - 10 * \left\lfloor \frac{j}{10} \right\rfloor$
        $j \leftarrow \left\lfloor \frac{j}{10} \right\rfloor$
    endwhile
endfor

Answer
For a given *i*, we find how many times does the *while* instruction runs

$$\Rightarrow complexity\ is\ \boldsymbol{\theta(n^2 \log_2 n)}$$

RECURSION

operation(*n, i*):
    if $n > 1$ then
        $i \leftarrow 2 * i$
        $m \leftarrow \left[\frac{n}{2}\right]$
        operation (*m, i* – 2)
        operation (*m, i* – 1)
        operation (*m, i* + 2)
        operation (*m, i* + 1)
    else
        write *i*
    endif
end

We mark with the time complexity of function ***operation*** with ***T(n)***. This being a recursive function, we write the recursive formula for calculating it as:

$$T(n) = \begin{cases} 1, & if\ n = 0 \\ 4T\left(\frac{n}{2}\right) + 1, & otherwise \end{cases}$$

$T(n) = 4T\left(\frac{n}{2}\right) + 1$ (1)

$T\left(\frac{n}{2}\right) = 4T\left(\frac{n}{4}\right) + 1$ (2)

$T\left(\frac{n}{4}\right) = 4T\left(\frac{n}{8}\right) + 1$ (3)

....

(1), (2) => $T(n) = 4T\left(\frac{n}{2}\right) + 1 = 4\left(4T\left(\frac{n}{4}\right) + 1\right) + 1 = 4^2T\left(\frac{n}{2^2}\right) + 4 \cdot 1 + 1$ (4)

(1), (4) => $T(n) = 4^2T\left(\frac{n}{2^2}\right) + 4 \cdot 1 + 1 = 4^2\left(4T\left(\frac{n}{8}\right) + 2\right) + 4 \cdot 1 + 1 = 4^3T\left(\frac{n}{2^3}\right) + 4^2 + 4 + 1$

We replace $n = 2^k$ (5) and by generalization we can write:

$T(n) = 4^kT\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 4^i = (2^k)^2T\left(\frac{n}{2^k}\right) + \frac{4^k-1}{4-1} = (2^k)^2T\left(\frac{n}{2^k}\right) + \frac{1}{3} \cdot ((2^k)^2 - 1)$ (6)

(5), (6) => $T(n) = n^2 + \frac{1}{3}(n^2 - 1)$ => $\boldsymbol{\theta(n^2)}$

BINARY SEARCH

```
def recbs(list, key, l, r):
    if r<l:
        return None
    m = (l + r) // 2
    if list[m]>key:
        return recbs(list, key, l, m-1)
    if list[m]<key:
        return recbs(list, key, m+1, r)
    if list[m]==key:
        return m
def iterbs(list, key):
    l = 0
    r = len(list)
```

```
    while l<=r:
        m = (l+r)/2
        if list[m]>key:
            r=m-1
        if list[m]<key:
            l=m+1
        if list[m]==key:
            return m
    return None
l = [1, 2, 4, 5, 6, 9, 10, 12]
```

T(n) = 1 + T(n/2)

$n = 2^k, T(2^k) = 1 + T(2^{k-1}) = 2 + T(2^{k-2}) = \cdots = k + kT(0). Since\ k = \log_2 n\ we\ have\ that\ T(n) = \mathbf{\log_2 n}$

SEARCHING. BUBBLE SORT

GENERATE SOME DATA FOR AVERAGE/WORST/BEST CASE

```
def avgcase(n):
    l = bestcase(n)
    random.shuffle(l)
    return l
def worstcase(n):
    l = []
    while n >0:
        l.append(n)
        n-=1
    return l
def bestcase(n):
    l = []
    while n>0:
        l.insert(0, n)
        n-=1
    return l
```

BUBBLE SORT IMPLEMENTATION

```
def bubblesort(l):
    done = False
    while done==False:
        done = True
        for i in range(0, len(l)-1):
            if l[i]>l[i+1]:
                a = l[i]
                l[i]=l[i+1]
                l[i+1]=a
                done = False
    return l

def optbubblesort(l):
    n = len(l)-1
    done = False
    while done==False:
        done = True
        for i in range(0, n):
            if l[i]>l[i+1]:
                a = l[i]
                l[i]=l[i+1]
                l[i+1]=a
                done = False
```

```
        n-=1
    return l
```

Run the code on your own computer. How does it fare when compared with the data below?

	Simple			Optimized		
N	**Best**	**Average**	**Worst**	**Best**	**Average**	**Worst**
1000	**0**	**0.45**	**0.60**	**0**	**0.31**	**0.45**
2000	**0.01**	**1.81**	**2.40**	**0**	**1.21**	**1.82**
3000	**0**	**4.07**	**5.45.**	**0.01**	**2.71**	**4.07**
4000	**0**	**7.18**	**9.70**	**0**	**4.82**	**7.31**
5000	**0**	**11.35**	**15.18**	**0**	**7.64**	**11.37**

Worst Case:

- to send the largest element (assume sorting is in increasing order) to its position, there are (n-1) comparisons and swaps
- for the second largest element, there are (n-2) comparisons and swaps
- the second smallest element will require a single comparison and swap

T(n)= (n-1) + (n-2) + ... + 1, a known sum that leads to $\boldsymbol{\theta}(\boldsymbol{n}^2)$

INSERT SORT

```python
def insertsort(data):
    for i in range(1, len(data)):
        val = data[i]
        j = i - 1
        while (j >= 0) and (data[j] > val):
            data[j + 1] = data[j]
            j = j - 1
        data[j + 1] = val
    return data
```

ASYMPTOTIC ANALYSIS

c1 = comparison + initial assignment, c2 = shift

Best Case: T(n) = n*c1 => $T(n) \in O(n)$

Worst Case: T(n) = n*c1+[n(n-1)/2]*c2 => $T(n) \in O(n^2)$

EMPIRICALLY

TABLE 1 - INSERT SORT RUNTIMES

	InsertSort		
N	**Best**	**Avg**	**Worst**
1000	0	0.15	0.28
2000	0	0.58	1.16
3000	0	1.30	2.66
4000	0	2.33	4.71
5000	0	3.65	7.33

MERGE SORT

```
def mergeSort(data):
    if len(data) > 1:
        mid = len(data) // 2
        leftHalf = data[:mid]
        rightHalf = data[mid:]

        mergeSort(leftHalf)
        mergeSort(rightHalf)

        merge(leftHalf, rightHalf, data)

def merge(data1, data2, result):
    i = 0
    j = 0
    aux = []

    while i < len(data1) and j < len(data2):
        if data1[i] < data2[j]:
            aux.append(data1[i])
            i = i + 1
        else:
            aux.append(data2[j])
            j = j + 1

    while i < len(data1):
        aux.append(data1[i])
        i = i + 1

    while j < len(data2):
        aux.append(data2[j])
        j = j + 1
    result.clear()
    result.extend(aux)
```

The complexity of the algorithm merging two lists of length m and n, respectively is $\boldsymbol{\theta(m+n)}$. Therefore, when sorting the two halves of a list of length n, the number of operations performed in the merge algorithm will be a linear function of n. We denote by *T(n)* the time complexity of the algorithm **mergeSort**. Given that this algorithm is calling itself, recursively, we can write the recursive formula for calculating *T(n)*:

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

$$T(n) = 2T\left(\frac{n}{2}\right) + n + C \qquad (1)$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2} + C \qquad (2)$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4} + C \qquad (3)$$

$$(1), (2) \Rightarrow T(n) = 2T\left(\frac{n}{2}\right) + n + C = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2} + C\right) + n + C = 2^2 T\left(\frac{n}{2^2}\right) + 2n + 2C + C \ (4)$$

(1), (3), (4) => $T(n) = 2^2T\left(\frac{n}{2^2}\right) + 2n + 2C + C \ = 2^2\left(2T\left(\frac{n}{8}\right) + \frac{n}{4} + C\right) + 2n + 2C + C = 2^3T\left(\frac{n}{2^3}\right) + \ 3n + C(2^2 + 2 + 1)$

By denoting $n \cong 2^k$ (5) and generalising the previous formulae, we ca write the following:

$T(n) = \ 2^kT\left(\frac{n}{2^k}\right) + kn + C \cdot \sum_{i=0}^{k-1} 2^i = 2^kT\left(\frac{n}{2^k}\right) + kn + C \cdot \frac{2^k-1}{2-1} = 2^kT\left(\frac{n}{2^k}\right) + kn + C \cdot (2^k - 1)$ (6)

(5) => $k \cong \log_2 n$ (7)

(6), (7) => $T(n) = \ n \cdot T(1) + n \cdot \log_2 n + nC = n \cdot (\log_2 n + C + 1)$ => *complexity* $\boldsymbol{\theta(n \log_2 n)}$