Lecture 9 – Recursion, Complexity

- ***Recursion***
- ***Complexity***

Recursion

A recursive definition (or inductive definition) is used to define an object in terms of itself.

A recursive definition of a function defines values of the functions for some inputs in terms of the values of the same function for other inputs.

```python
def factorial(n):
    """
      compute the factorial
      n  is a positive integer
      return n!
    """
    if n== 0:
        return 1
    return  factorial(n-1)*n
```

- Direct recursion : P invoke P
- Indirect recursion P invoke Q, Q invoke P

Main idea:

- base case: simplest possible solution
- inductive step: break the problem into a simpler version of the same problem plus some other steps

```python
def recursiveSum(l):
    """
    Compute the sum of numbers
    l - list of number
    return int, the sum of numbers
    """
    #base case
    if l==[]:
        return 0
    #inductive step
    return l[0]+recursiveSum(l[1:])
```

```python
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

Obs recursiveSum(l[1:]):

l[1:] - is creating a copy of the list

exercise: modify the recursiveSum to avoid l[1:]

How recursion works:

- on each method invocation a new symbol table is created. The symbol table contains all the parameters and the local variables defined in the function
- the symbol tables are stored in a stack, when a function is returning the current symbol tale is removed from the stack

```
def isPalindrome(str):
    """
      verify if a string is a palindrome
      str - string
      return True if the string is a palindrome False otherwise
    """
    dict = locals()
    print id(dict)
    print dict

    if len(str)==0 or len(str)==1:
        return True

    return str[0]==str[-1] and isPalindrome(str[1:-1])
```

Recursion

Advantages:

- clarity
- simplified code

Disadvantages:

- memory consumption for large recursion depth
    - For each recursion a new symbol table is created

Computational complexity

Concerned with studying the algorithms efficiency.

We compare algorithms with respect to:

- the *amount of necessary space* to hold temporary data,
- the computing speed, i.e. the *running-time* necessary to solve the problem.

program running-time is the time necessary for a program to run.

Depends on:

- the input data
- the changes from a run to another
- the used hardware.

Running time example

```
def fibonacci(n):
    """
     compute the fibonacci number
     n - a positive integer
     return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

```
def fibonacci2(n):
    """
     compute the fibonacci number
     n - a positive integer
     return the fibonacci number for a given n
    """
    sum1 = 1
    sum2 = 1
    rez = 0
    for i in range(2, n+1):
        rez = sum1+sum2
        sum1 = sum2
        sum2 = rez
    return rez
```

```
def measureFibo(nr):
    sw = StopWatch()
    print "fibonacci2(", nr, ") =", fibonacci2(nr)
    print "fibonacci2 take " +str(sw.stop())+" seconds"

    sw = StopWatch()
    print "fibonacci(", nr, ") =", fibonacci(nr)
    print "fibonacci take " +str(sw.stop())+" seconds"


measureFibo(32)
```

```
fibonacci2( 32 ) = 3524578
fibonacci2 take 0.0 seconds
fibonacci( 32 ) = 3524578
fibonacci take 1.7610001564 seconds
```

Efficiency of a function

- the amount of resources they use, usually measured in either the *space* or *time* used.

Measuring efficiency:

- a mathematical analysis , called ***asymptotic analysis***
  can capture aspects of efficiency for all possible inputs but not exact execution times.
- an ***empirical analysis***
  determine exact running times for a sample of specific inputs,
  cannot predict the performance of the algorithm on all inputs.

Running time of an algorithm is studied in direct relation to the size of input data.

- Estimate the running time of an algorithm for a specific, stated size input data .
- We are focusing on ***asymptotic analysis***

Complexity

- **best case** - for the data set leading to the minimum running time
    - *best-case complexity* (BC): $BC(A) = \min_{I \in D} E(I)$
- **worst case**, for the data set leading to the maximum running time.
    - *worst-case complexity* (WC): $WC(A) = \max_{I \in D} E(I)$
- **average** running time of an algorithm.
    - *average complexity* (AC): $AC(A) = \sum_{I \in D} P(I)E(I)$

A - algorithm; D domain of algorithm this algorithm for inputs of size *n; E(I)* number of operations performed ; *P(I)* the probability of having *I* as input data of the algorithm

Capture the essence: how the running time of an algorithm increases with the size of the input *at the limit*

(if $n \to \infty$, then $3 \cdot n^2 \approx n^2$).

Compare algorithms by using the *magnitude order* of their running-time complexity

Running time complexity

- running time is not a fixed number, but rather a function of the input data size *n*, denoted *T*(*n*).
- measure basic "steps" that the algorithm makes (for example, the number of statements executed).
- will not exactly predict the true running
- it will get us within a small constant factor of the true running time most of the time.

Example : $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$

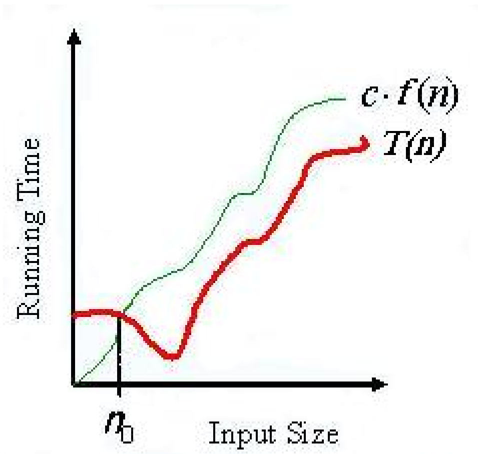
Because $0 < \log_2 n < n,\ \forall n > 1$ and $\sqrt{n} < n,\ \forall n > 1$, we can conclude that n^3 term dominates for large *n*

So, as a conclusion, we can say that the running time *T*(*n*) grows "roughly on the order of n^3", and this is written $T(n) \in O(n^3)$.

Informally, the statement $T(n) \in O(n^3)$ means, "when you ignore constant multiplicative factors, and consider the leading (i.e. fastest growing) term, you get n^3".

We will denote by f a function $f : N \to \Re$ and by T the function that gives the execution time of an algorithm, $T : N \to N$.

Definition 1. ("Big-oh", O-notation). We say that $T(n) \in O(f(n))$ if exist **c** and $\mathbf{n_0}$ positive constants (independent of n) such that $0 \le T(n) \le c \cdot f(n), \quad \forall n \ge n_0$.



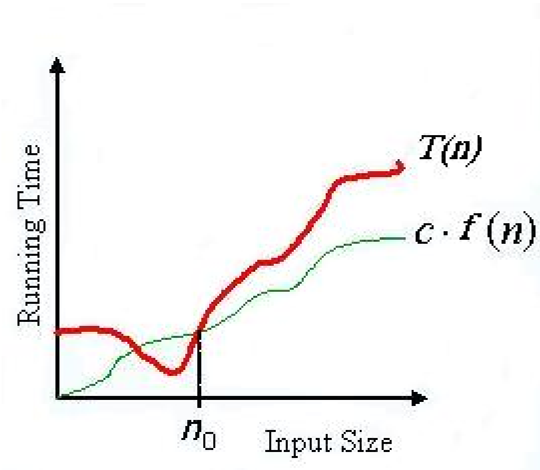
In other words, O notation gives the asymptotic upper bound

Alternative definition 1. We say that $T(n) \in O(f(n))$ if $\lim_{n\to\infty}\frac{T(n)}{f(n)}$ is 0 or is a constant, but **not** ∞

Remarks.

1. If $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$, then $\lim_{n\to\infty}\frac{T(n)}{n^3} = 13$. So, we can say that $T(n) \in O(n^3)$.
2. The O notation is good for putting an upper bound on a function. We notice that if $T(n) \in O(n^3)$, it is also $O(n^4)$, $O(n^5)$, etc since the limit will just go to zero. That is why we will need a notation for the lower bound of the complexity. This notation is Ω.

Definition 2. ("Big-omega", Ω-notation). We say that $T(n) \in \Omega(f(n))$ if exist **c** and **n_0** positive constants (independent of n) such that $0 \le c \cdot f(n) \le T(n), \quad \forall n \ge n_0$.



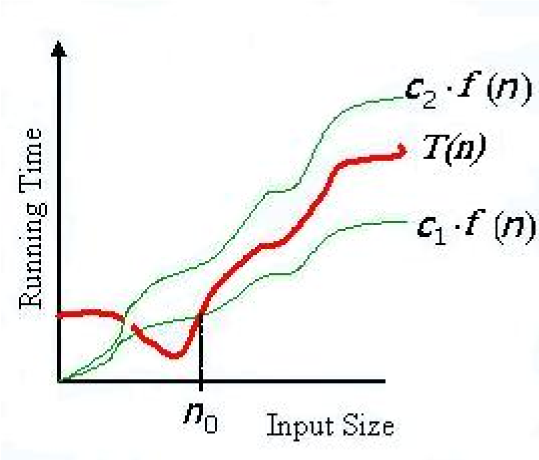
In other words, Ω notation gives the asymptotic lower bound

Alternative definition 2. We say that $T(n) \in \Omega(f(n))$ if $\lim_{n\to\infty} \frac{T(n)}{f(n)}$ is a constant or ∞, but **not** 0.

Remark: If $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$, then $\lim_{n\to\infty} \frac{T(n)}{n^3} = 13$. So, we can say that $T(n) \in \Omega(n^3)$, also.

Definition 3. ("Big-theta", θ-notation). We say that $T(n) \in \theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$, i.e., exist **c1, c2** and $\mathbf{n_0}$ positive constants (independent of n) such that

$$c1 \cdot f(n) \leq T(n) \leq c2 \cdot f(n), \qquad \forall n \geq n_0.$$



In other words, θ notation gives the asymptotic tight bound.

Alternative definition 3. We say that $T(n) \in \theta(f(n))$ if $\lim_{n\to\infty} \frac{T(n)}{f(n)}$ is a constant (but **not** 0 or ∞).

Remarks.

1. The running time of an algorithm is $\theta(f(n))$ if and only if its worst case running time is $O(f(n))$ and its best case running time is $\Omega(f(n))$.
2. Notation $O(f(n))$ is often misused instead of $\theta(f(n))$.
3. If $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$, then $\lim_{n\to\infty} \frac{T(n)}{n^3} = 13$. So, $T(n) \in \theta(n^3)$. This can also be deduced from $T(n) \in O(n^3)$ and $T(n) \in \Omega(n^3)$.

Summations

for i *in range(0, n):*

#some instructions

Assuming that the loop body (the *) takes $f(i)$ time to run, the total running time is given by the summation

$$T(n) = \sum_{i=1}^{n} f(i)$$

We can observe that nested loops naturally lead to nested sums.

Solving summations breaks down into two basic steps

- simplify the summation as much as possible -by removing constant terms and separating individual terms into separate summations.
- each of the remaining simplified sums can be solved.

Summation Examples

Analyze the time complexity of the following functions

<table>
<tr><td>

```python
def f1(n):
    s = 0
    for i in range(1,n+1):
        s=s+i
    return s
```

</td><td>

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

Overall complexity $\Theta(n)$

Best/Average/Worst case is the same

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

```python
def f2(n):
    i = 0
    while i<=n:
        #atomic operation
        i = i + 1
```

</td><td>

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

Overall complexity $\Theta(n)$

Best/Average/Worst case is the same

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

```python
def f3(l):
   """
   l - list of numbers
   return True if the list contains
an even nr
   """
   poz = 0
   while poz<len(l) and l[poz]%2 !=0:
       poz = poz+1
   return poz<len(l)
```

</td><td>

Best case:

The first element is an even number: $T(n)=1\in\Theta(1)$

Worst case: No even number in the list: $T(n)=n\in\Theta(n)$

Average Case:

While can be executed 1,2,..n times (same probability).

Number of steps = the average number of while iterations

$T(n)=(1+2+\ldots+n)/n=(n+1)/2 \rightarrow T(n)\in\Theta(n)$

Overall complexity $O(n)$

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

Summation Examples

<pre>def f4(n):
    for i in range(1,2*n-2):
        for j in range(i+2,2*n):
            #some computation
            pass</pre>
	$T(n)=\sum_{(i=1)}^{(2n-2)} \sum_{(j=i+2)}^{2n} 1=\sum_{(i=1)}^{(2n-2)} (2n-i-1)$
	
	$T(n)=\sum_{(i=1)}^{(2n-2)} 2n-\sum_{(i=1)}^{(2n-2)} i-\sum_{(i=1)}^{(2n-2)} 1$
	
	$T(n)=2n\sum_{(i=1)}^{(2n-2)} 1-(2n-2)(2n-1)/2-(2n-2)$
	
	...
	
	$T(n)=2n^2-3n+1\in\Theta(n^2)$ Overall complexity $\Theta(n^2)$

<pre>def f5():
    for i in range(1,2*n-2):
        j = i+1
        cond = True
        while j<2*n and cond:
            #elementary operation
            if someCond:
                cond = False</pre>
	Best case: While executed once
	$T(n)=\sum_{(i=1)}^{(2n-2)} 1=2n-2\in\Theta(n)$
	Worst case: While executed $2n-(i+1)$ times
	$T(n)=\sum_{(i=1)}^{(2n-2)} (2n-i-1)=...=2n^2-3n+1\in\Theta(n^2)$
	Average case:
	For a fixed I the While can be executed 1,2..2n-i-1 times
	average steps : $C_i=(1+2+...+2n-i-1)/2n-i-1=...=(2n-i)/2$
	
	$T(n)=\sum_{(i=1)}^{(2n-2)} C_i=\sum_{(i=1)}^{(2n-2)} (2n-i)/2=...\in\Theta(n^2)$
	
	Overall complexity $O(n^2)$

Some important sums to know are:

$$\sum_{i=1}^{n} 1 = n$$ The constant series.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$ The arithmetic series.

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{2}$$ The quadratic series.

$$\sum_{i=1}^{n} \frac{1}{i} = \ln(n) + O(1)$$ The harmonic series.

$$\sum_{i=1}^{n} c^i = \frac{c^{n+1} - 1}{c - 1}, \quad c \neq 1$$ The geometric series.

As can be seen above, geometric progressions exhibit exponential growth.

Common complexities

$T(n) \in O(1)$ - ***constant* time.** It is a great complexity. This means that the algorithm takes only constant time.

$T(n) \in O(\log_2 \log_2 n)$ - it is a very fast time (as fast as a constant time)

$T(n) \in O(\log_2 n)$ - it is a very good time. This is called ***logarithmic*** time. It is the running time of binary search and the height of a balanced binary tree. This is about the best that can be achieved for data structures based on binary trees.

We note that $\log_2 1000 \approx 10, \quad \log_2 1.000.000 \approx 20$.

$T(n) \in O((\log_2 n)^k)$ - (where k is a constant). This is called ***polylogarithmic*** time**.** It is not bad, when simple logarithmic time is not achievable.

Common complexities

$T(n) \in O(n)$ **-** This is called ***linear*** time. It is about the best that one can hope for an algorithm that has to look at all the data.

$T(n) \in O(n \cdot \log_2 n)$ - This one is famous, because this is the time needed to sort a list of numbers (Merge-Sort, Qiuck-Sort). It arises in a number of other problems as well.

$T(n) \in O(n^2)$ **-** ***Quadratic*** time. Okay if n is in the thousands, but rough when n gets into the millions.

$T(n) \in O(n^k)$ **-** (where k is a constant). This is called ***polynomial*** time. Practical if k is not too large.

$T(n) \in O(2^n), O(n^3), O(n!)$ **-** *Exponential* **time.** Algorithms having this time complexity are only practical for small values of n **:** $n \le 10, n \le 20$.

Recurrences

A **recurrence** is a mathematical formula that is defined recursively.

For example, let us consider the previous problem of determining the number $N(h)$ of nodes of a 3-ary tree of height h. By a simple analysis, we can observe that $N(h)$ can be described using the following recurrence:

$$\begin{cases} N(0) = 1 \\ N(h) = 3 \cdot N(h-1) + 1, \qquad h \geq 1 \end{cases}$$

The explanation is given below:

- The number of nodes of a complete 3-ary tree of height 0 is 1.
- A complete 3-ary tree of height h (h>0) consists of a root node and 3 copies of a 3-ary tree of height h-1.

If we solve the above recurrence, we obtain that:

$$N(h) = 3^h \cdot N(0) + (1 + 3^1 + 3^2 + \ldots + 3^{h-1}) = \sum_{i=0}^{h} 3^i \ ,$$

the same result obtained by computing $N(h)$ using summations, not recurrences.

Recurrence example

<pre>def **recursiveSum**(l):
    """
    Compute the sum of numbers
    l - list of number
    return int, the sum of numbers
    """
    #base case
    if l==[]:
        return 0
    #inductive step
    return l[0]+recursiveSum(l[1:])</pre>
	Recurrence: $T(n)=\begin{cases} 1 \; for \; n=0 \\ T(n-1)+1 \; otherwise \end{cases}$
	
	$T(n)=T(n-1)+1$
	$T(n-1)=T(n-2)+1$
	$T(n-2)=T(n-3)+1$ => $T(n)=n+1 \in \Theta(n)$
	$\ldots = \ldots$
	$T(1)=T(0)+1$

<pre>def **hanoi**(n, x, y, z):
    """
       n -number of disk on the x
stick
       x - source stick
       y - destination stick
       z - intermediate stick
    """
    if n==1:
      print "disk 1 from",x,"to",y
      return
    hanoi(n-1, x, z, y)
    print "disk ",n,"from",x,"to",y
    hanoi(n-1, z, y, x)</pre>
	Recurrence: $T(n)=\begin{cases} 1 \; for \; n=1 \\ 2\mathrm{T}(n-1)+1 \; otherwise \end{cases}$
	
	$T(n)=2T(n-1)+1$
	$T(n-1)=2T(n-2)+1=$
	$T(n-2)=2T(n-3)+1$
	$\ldots = \ldots$
	$T(1)=T(0)+1$
	
	=>
	
	$T(n)=2T(n-1)+1$
	$2T(n-1)=2^2T(n-2)+2$
	$2^2T(n-2)=2^3T(n-3)+2^2$
	$\ldots = \ldots$
	$2^{(n-2)}T(2)=2^{(n-1)}T(1)+2^{(n-2)}$
	
	$T(n)=2^{(n-1)}+1+2+2^2+2^3+\ldots+2^{(n-2)}$
	
	$T(n)=2^n-1 \in \Theta(2^n)$

Space complexity

The ***space*** **complexity** of an algorithm estimates the quantity of memory space required by the algorithm to store the input data, the final results and the intermediate results. As the *time* complexity, the *space* complexity is also estimated using O, Θ, Ω notations.

All the remarks from related to the asymptotic notations used in running time complexity analysis are valid for the space complexity, also.

Space complexity example

<pre>def iterativeSum(l):
    """
    Compute the sum of numbers
    l - list of number
    return int, the sum of numbers
    """
    rez = 0
    for nr in l:
        rez = rez+nr
    return rez</pre>
	We need space to store the numbers
	
	$T(n)=n \in \Theta(n)$

<pre>def recursiveSum(l):
    """
    Compute the sum of numbers
    l - list of number
    return int, the sum of numbers
    """
    #base case
    if l==[]:
        return 0
    #inductive step
    return l[0]+recursiveSum(l[1:])</pre>
	Recurrence: $T(n)=\begin{cases} 0 \ for \ n=1 \\ T(n-1)+n-1 \ otherwise \end{cases}$

Time/space complexity for a function - overview

1 If there is Best/Worst case:

- describe **Best case**
- compute complexity for **Best Case**
- describe **Worst Case**
- compute complexity for **Worst case**
- compute **average** complexity
- compute **overall** complexity

2 If Best = Worst = Average

- compute complexity

Compute complexity:

- if we have a **recurrence**:
  - compute using equalities
- else
  - compute using **summations**