Recursion. Computational complexity

Lect. PhD. Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

Overview

Second Laboratory Test

- Will take place week 12, during the laboratory
- You will receive one problem statement, from what was studied for Assignment 5-9
- You can use your own laptop, with an empty workspace
- Only documentation is **offline** Python documentation
- Grading will be during the laboratory, grade is 30% of final lab grade

Recursion

Circular definition

In order to understand recursion, one must first understand recursion.

What is recursion?

- A recursive 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.
- Recursion can be:
  - **Direct** - a function **p** calls itself
  - **Indirect** - a function **p** calls another function, but it will be called again in turn

Demo

Recursion

Examine the source code in **ex30_recursion.py**

Recursion

- **Main idea**
  - Base case: simplest possible solution
  - Inductive step: break the problem into a simpler version of the same problem plus some other steps
- **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
- **Recursion and stack memory**
  - Stack memory size is allocated by the compiler/runtime environment
  - Compilers can optimize recursive computation (e.g. see Ackermann's function)

Recursion

- **Advantages**
  - + Clarity
  - + Simplified code
- **Disadvantages**
  - – Large recursion depth might run out of stack memory
  - – Large memory consumption in the case of branched recursive calls (for each recursion a new symbol table is created - see Ackermann's function)

Computational complexity

What is it?

Studying algorithm efficiency mathematically

- **We study algorithms with respect to**
  - Run time required to solve the problem
  - Extra memory required for temporary data
- **What affects runtime for a given algorithm**
  - Size and structure of the input data
  - Hardware
  - Changes from a run to another due to hardware and software environment

Running time example

As a first example, lets take a well-understood function: computing the nth term of the Fibonacci sequence

- What is so special about it?
  - Easy to write in most programming languages
  - Iterative and recursive implementation comes naturally
  - Different run-time complexity!

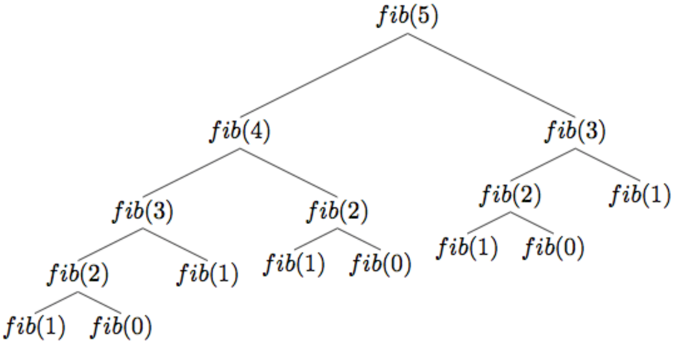
Demo

Computational complexity

Examine the source code in **ex31_complexity.py**[1]

[1]To run the example, install the texttable component from https://github.com/foutaise/texttable

Overcalculation in recursive Fibonacci



Demo

Discussion

How can overcalculation be eliminated?

Memoization

Examine the source code in **ex32_complexityOptimized.py**[2]

[2] To run the example, install the texttable component from https://github.com/foutaise/texttable

Efficiency of a function

What is function efficiency?

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

- **Measuring efficiency**
  - **Asymptotic analysis** - mathematical analysis that captures efficiency aspects for all possible inputs but cannot provide execution times.
  - **Empirical analysis** - determines exact running times for a sample of specific inputs, but cannot predict algorithm performance on all inputs.
- **Function run time is studied in direct relation to data input size**
- We focus on asymptotic analysis, and illustrate it using empirical data.

Complexity

- **Best case (BC)**, for the data set leading to minimum running time $\text{BC(A)} = \min_{I \in D} E(I)$
- **Worst case (WC)**, for the data set leading to maximum running time $\text{WC(A)} = \max_{I \in D} E(I)$
- **Average case (AC)**, average running time of the algorithm $\text{AC(A)} = \sum_{I \in D} P(I)E(I)$

Legend

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

Complexity

Observation

Due to the presence of the **P(I)** parameter, calculating average complexity might be challenging

Run time complexity

The essence

- How the running time of an algorithm increases with the size of the input at the limit: **if** $n \to \infty$**, then** $3n^2 \approx n^2$
- We compare algorithms using the magnitude order of their runtime complexity

Run 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 (e.g. number of statements executed).

\+ It gets us within a small constant factor of the true runtime most of the time.

\+ Allows us to predict run time for different input data

− Does not exactly predict true runtime

Run time complexity

- **Example:**

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

- Because $0 < \log_2 n < n, \forall n > 1$, and $\sqrt{n} < n, \forall n > 1$, we conclude that the n^3 term dominates for large n.
- Therefore, we say that the running time $T(n)$ grows *"roughly on the order of n^3"*, and we write it as $T(n) \in O(n^3)$.
- Informally, the statement above means that *"when you ignore constant multiplicative factors, and consider the leading term, you get n^3"*.

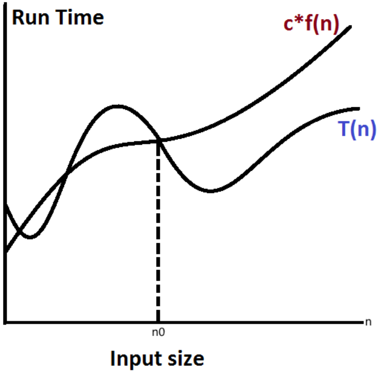
"Big-O" notation

We denote function $f : \mathbb{N} -> \mathbb{R}$, and by T the function that gives the execution time of an algorithm, $T : \mathbb{N} -> \mathbb{N}$.

Definition, "Big-oh" notation

We say that $T(n) \in O(f(n))$ if there exist c and n_0 positive constants independent of n such that $0 \leq T(n) \leq c * f(n), \forall n \geq n_0$.

"Big-O" notation



- In other words, $O(n)$ notation provides the asymptotic upper bound.

"Big-O" notation

Alternative definition, "Big-oh" notation

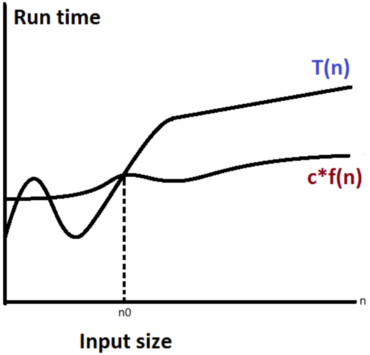
We say that $T(n) \in O(f(n))$ if $\lim_{n->\infty} \frac{T(n)}{f(n)}$ is 0 or a constant, but not ∞.

- If $T(n) = 13 * n^3 + 42 * n^2 + 2 * n * \log_2 n + 3 * \sqrt{n}$, then $\lim_{n->\infty} \frac{T(n)}{f(n)} = 13$. So, we say that $T(n) \in O(n^3)$.
- 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 $O(n^4)$, $O(n^5)$, since the limit will go to 0. To be more precise, we will also introduce a lower bound on complexity.

"Big-omega" notation

Definition, "Big-omega" notation

We say that $T(n) \in \Omega(f(n))$ if there exist c and n_0 positive constants independent of n such that $0 \leq c * f(n) \leq T(n), \forall n \geq n_0$.



"Big-omega" notation

Alternative definition, "Big-omega" notation

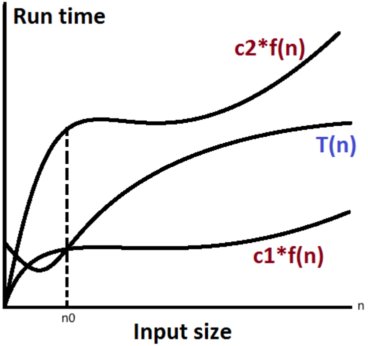
We say that $T(n) \in \Omega(f(n))$ if $\lim_{n->\infty} \frac{T(n)}{f(n)}$ is a constant or ∞, but not 0.

- If $T(n) = 13 * n^3 + 42 * n^2 + 2 * n * \log_2 n + 3 * \sqrt{n}$, then $\lim_{n->\infty} \frac{T(n)}{f(n)} = 13$. So, we say that $T(n) \in \Omega(n^3)$.
- The Ω notation is used for putting a lower bound on a function.

"Big-theta" notation

Definition, "Big-theta" 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. there exist c_1, c_2 and n_0 positive constants, independent of n such that $c_1 * f(n) \leq T(n) \leq c_2 * f(n), \forall n \geq n_0$.



"Big-theta" notation

Alternative definition, "Big-theta" notation

We say that $T(n) \in \Theta(f(n))$ if $\lim_{n->\infty} \frac{T(n)}{f(n)}$ is a constant (but not 0 or ∞).

- If $T(n) = 13 * n^3 + 42 * n^2 + 2 * n * \log_2 n + 3 * \sqrt{n}$, then $\lim_{n->\infty} \frac{T(n)}{f(n)} = 13$. So, we say that $T(n) \in \Theta(n^3)$. This can also be deduced from $T(n) \in O(n^3)$ and $T(n) \in \Omega(n^3)$
- The run time of an algorithm is $\Theta(f(n))$ if and only if its worst case run time is $O(f(n))$ and best case run time is $\Omega(f(n))$.

Summations


```
for i in dataList:
    # do somthing here...
```

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

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

Observation

Nested loops naturally lead to nested sums.

Summation

Solving summations breaks down into two basic steps

- Simplify the summation as much as possible - remove constant terms and separate individual terms into separate summations.
- Solve each of the remaining simplified sums.

Summation - examples


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

- $T(n) = \sum_{i=1}^{n} 1 = n \Rightarrow T(n) \in \Theta(n)$
- BC/AC/WC complexity is the same

Summation - examples


```
def f(n):
    i = 0
    while i <= n:
        # do something here...
        i += 1
```

- $T(n) = \sum_{i=1}^{n} 1 = n \Rightarrow T(n) \in \Theta(n)$
- BC/AC/WC complexity is the same

Summation - examples

```
def f(l):
    '''
    Return True if list contains an even number
    '''
    poz = 0
    while poz < len(l) and l[poz]%2 !=0:
        poz += 1
    return poz<len(l)
```

- BC - first element is even number, $T(n) = 1$, $T(n) \in \Theta(1)$
- WC - no even number in list, $T(n) = n$, $T(n) \in \Theta(n)$

Summation - examples


```python
def f(l):
    '''
    Return True if list contains an even number
    '''
    poz = 0
    while poz < len(l) and l[poz]%2 !=0:
        poz += 1
    return poz<len(l)
```

- AC - the **while** can be executed $1, 2, \ldots n$ times, with same probability (lacking additional information). The number of steps is then the average number of iterations: $T(n) = \frac{1+2+..+n}{n} = \frac{n+1}{2} \Rightarrow T(n) \in \Theta(n)$

Summation - examples

```
def f(n):
    for i in range(1,2*n-2):
        for j in range(i+2,2*n):
            # do something...
```

- $T(n) = \sum_{i=1}^{2n-2} \sum_{j=i+2}^{2n} 1 = \sum_{i=1}^{2n-2} (2n - i - 1)$
- $T(n) = \sum_{i=1}^{2n-2} 2n - \sum_{i=1}^{2n-2} i - \sum_{i=1}^{2n-2} 1$
- $T(n) = 2n * \sum_{i=1}^{2n-2} 1 - \frac{(2n-2)(2n-1)}{2} - (2n-2)$
- $T(n) = 2 * n^2 - 3 * n + 1 \in \Theta(n^2)$.

Summation - examples


```
def f():
    for i in range(1,2*n-2):
        j = i+1
        cond = True
        while j<2*n and cond:
            # do something ...
            if someCondition:
                cond = False
```

- 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,

$$T(n) = \sum_{i=1}^{2n-2} (2n - i - 1) = \ldots = 2n^2 - 3n + 1 \in \Theta(n^2)$$

Summation - examples


```
def f():
    for i in range(1,2*n-2):
        j = i+1
        cond = True
        while j<2*n and cond:
            # do something ...
            if someCondition:
                cond = False
```

- Average Case - for a given i the "while" loop can be executed $1, 2, ..., 2n-i-1$ times, average steps: $c_i = \frac{1+2+...+2n-i-1}{2n} - i - 1 = ... = \frac{2n-i}{2}$
- $$T(n) = \sum_{i=1}^{2n-2} c_i = \sum_{i=1}^{2n-2} \frac{2n-i}{2} = ... \in \Theta(n^2)$$
- Overall complexity is therefore $\Theta(n^2)$

Summation - important sums

- Constant series $\sum_{i=1}^{n} 1 = n$
- Arithmetic series $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- Quadratic series $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{2}$
- Harmonic series $\sum_{i=1}^{n} \frac{1}{i} = \ln(n) + O(1)$
- Geometric series $\sum_{i=1}^{n} c^i = \frac{c^{n+1} - 1}{c - 1}, c \neq 1$

Common complexities

- **Constant time**: $T(n) \in O(1)$. It means that run time does not depend on size of the input. It is very good complexity.
- $T(n) \in O(\log_2 \log_2 n)$. This is also a very fast time, it is practically as fast as constant time.
- **Logarithmic time**: $T(n) \in O(\log_2 n)$. It is the run time of binary search and height of balanced binary trees. About the best that can be achieved for data structures using binary trees. Note that $\log_2 1000 \approx 10$, $\log_2 1000^2 \approx 20$.

Common complexities

- **Polylogarithmic time**: $T(n) \in O((\log_2 n)^k)$.
- **Liniar time**: $T(n) \in O(n)$. It means that run time scales liniarly with the size of input data.
- $T(n) \in O(n * \log_2 n)$. This is encountered for fast sort algorithms, such as merge-sort and quick-sort.

Common complexities

- **Quadratic time**: $T(n) \in O(n^2)$. Empirically, ok with n in the hundreds but not with n in the millions.
- **Polynomial time**: $T(n) \in O(n^k)$. Empirically practical when k is not too large.
- **Exponential time**: $T(n) \in O(2^n), O(n!)$. Empirically usable only for small values of input.

Recurrences

What is a recurrence?

A recurrence is a mathematical formula defined recursively.

Example I - Node count of complete 3-ary tree

- A **recurrence** is a mathematical formula that is defined recursively.
- For example, let us consider the problem of determining the number $N(h)$ of nodes of a complete 3-ary tree of height h. We can observe that $N(h)$ can be described using the following recurrence:

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

Example I - Node count of complete 3-ary tree

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:

$$N(h) = 3^h * N(0) + (1 + 3^1 + 3^2 + ... + 3^{h-1}) = \sum_{i=0}^{h} 3^i.$$

Example II - Recursive list summation

```
def recursiveSum(l):
    '''
    Compute the sum of numbers in a list
    l - input list
    return int, the sum of the numbers
    '''
    # base case
    if l == []:
        return 0
    # recursion step
    return l[0] + recursiveSum(l[1:])
```

- In this case, the reccurence is:

$$T(n) = \begin{cases} 1, n = 0 \\ T(n-1) + 1, n > 0 \end{cases}$$

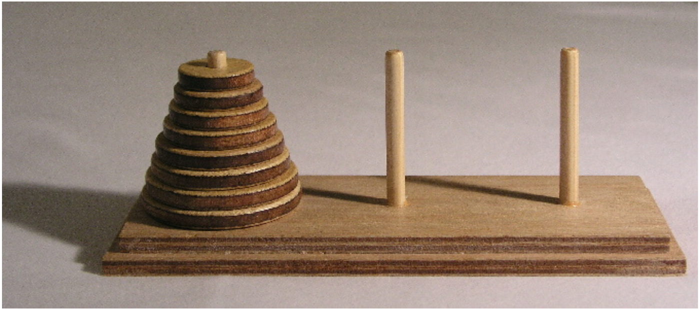
Example II - Recursive list summation

- Solving the reccurence:
  $$\mathrm{T(n)} = \begin{cases} 1, n = 0 \\ T(n-1) + 1, n > 0 \end{cases}$$
- $T(n) = T(n-1) + 1$
- $T(n-1) = T(n-2) + 1$
- $T(n-2) = T(n-3) + 1 \Rightarrow T(n) = n + 1 \in \Theta(n)$

Example III - Tower of Hanoi

Legend says there is an Indian temple containing a large room with three posts and surrounded by 64 golden discs. Brahmin priests, acting out an ancient prophecy, are moving these discs since time immemorial, according to the rules of the Brahma. According to the legend, when the last move is completed, **the world will end**.

Example III - Tower of Hanoi



Example III - Tower of Hanoi

A mathematical game. Starts with three rods and a number of discs of increasing radius placed on one of them. The objective of the game is to move all the discs to another rod, observing the following rules:

- You can only move one disk at a time
- You can only move the uppermost disc from a rod
- You cannot place a larger disc on a smaller one

Example III - Tower of Hanoi

So ... are we safe (for now)? Let's study this:

- **Mathematically**
- Empirically

Example III - Tower of Hanoi

The idea of the algorithm (for **n** discs):

- Move **n-1** discs from source to intermediate stick
- Move the last disc to the destination stick
- Solve problem for **n-1** discs

Example III - Tower of Hanoi

```
def hanoi(n, x, y, z):
    '''
    n - number of disks 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)
```

- The recurrence is:

$$T(n) = \begin{cases} 1, n = 1 \\ 2T(n-1) + 1, n > 1 \end{cases}$$

Example III - Tower of Hanoi

- Solving the recurrence:
  T(n) = $\begin{cases} 1, n = 1 \\ 2T(n-1)+1, n > 1 \end{cases}$
- $T(n) = 2T(n-1)+1,\ T(n-1) = 2T(n-2)+1,$ $T(n-2) = 2T(n-3)+1, \ldots,\ T(1) = T(0)+1$
- $T(n) = 2T(n-1)+1$, $2T(n-1) = 2^2T(n-2)+2$, $2^2T(n-2) = 2^3T(n-3)+2^2$,...,$2^{n-2}T(2) = 2^{n-1}T(1)+2^{n-2}$
- We have $T(n) = 2^{n-1} + 2^0 + 2^1 + 2^2 + \ldots + 2^{n-2}$
- Therefore $T(n) = 2^n - 1 \in \Theta(2^n)$

Example III - Tower of Hanoi

So ... are we safe for now? Let's study this:

- Mathematically
- **Empirically**

Demo

Recursion

Examine the source code in **ex33_hanoi.py**

Space complexity

What is the space complexity of an algorithm?

The space complexity estimates the quantity of memory 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" and "Omega" notation.

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

Example I - List summation


```
def iterativeSum(l):
    '''
    Compute the sum of numbers in a list
    l - input list
    return int, the sum of the numbers
    '''
    res = 0
    for nr in l:
        rez += nr
    return rez
```

- We need memory to store the numbers, so $T(n) = n \in \Theta(n)$.

Example I - List summation

```python
def recursiveSum(l):
    '''
    Compute the sum of numbers in a list
    l - input list
    return int, the sum of the numbers
    '''
    # base case
    if l == []:
        return 0
    # recursion step
    return l[0] + recursiveSum(l[1:])
```

- The recurrence is:

$$T(n) = \begin{cases} 0, n = 1 \\ T(n-1) + n - 1, n > 1 \end{cases}$$

Complexity 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 (if possible)
   - Compute overall complexity (if possible)
2. If Best $=$ Worst $=$ Average
   - Compute complexity