Recursion. Computational complexity

Assoc. Prof. Arthur Molnar

Babeș-Bolyai University

2024

Overview

Recursion

Circular definition

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

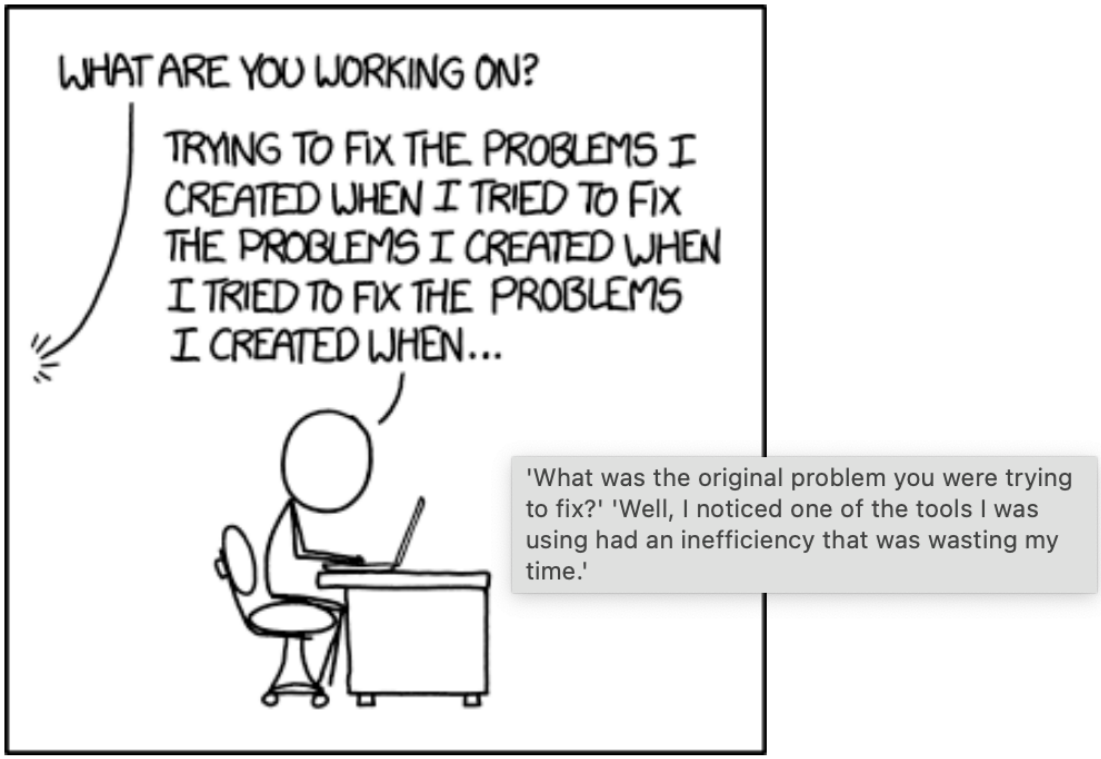


Figure: Source: `https://www.xkcd.com/1739/`

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 examples

ex04_recursion.py[a]

[a] ProTip: This is a clickable link

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[1]. 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

[1] You can check them via the builtin locals() and globals() functions

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 the Ackermann function)

In more detail...

Other resources

- https://realpython.com/python-recursion/
- https://realpython.com/python-thinking-recursively/

Optimizing recursive calls

Tail Call Optimization

- Not all language/compiler combos support it (Python, Java do not, C/C++ generally do)
- When the last thing the function does before returning is call itself - in this case, there is no functional need to build and keep a stack frame

\+ Transforms an explicit recursive call into an implicit iterative one

– Can make debugging and error reporting more difficult

Optimizing recursive calls

Tail Call Optimization - additional reading

- Hands-on explanations
  `https://dev.to/rohit/demystifying-tail-call-optimization-5bf3`
  `https://tratt.net/laurie/blog/2004/tail_call_optimization.html`
- Python does not support tail call optimization
  `http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html` – authored by Guido van Rossum, the creator of the Python language

Computational complexity

What is it?

Studying algorithm efficiency mathematically

- **We study algorithms with respect to**
  - Run time required to solve the problem
  - Extra memory-space required for temporary data
- **What affects the runtime of 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 n^{th} 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!
  - Tail call optimization can be applied in language/compiler combos that support it

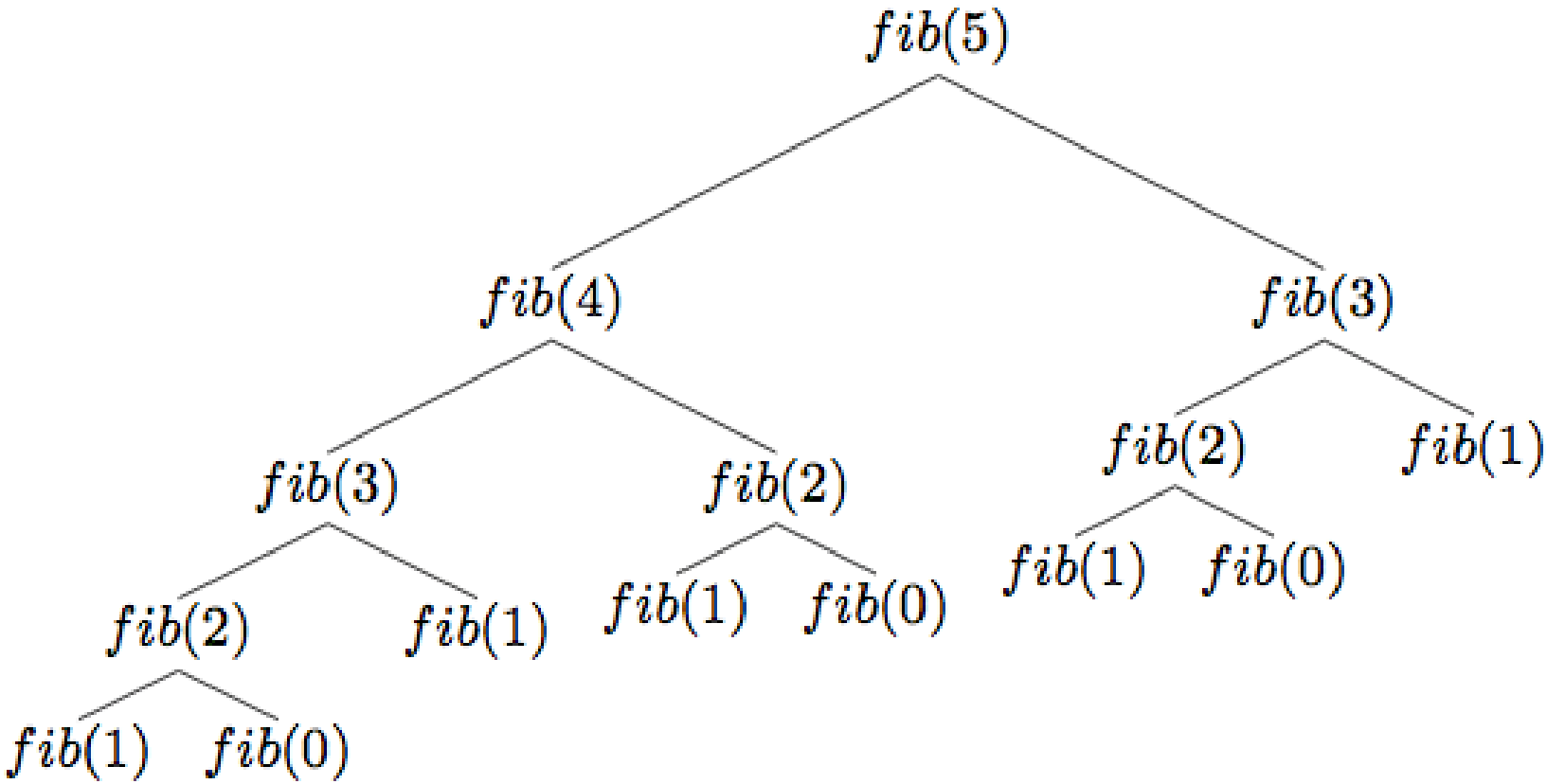
Demo

Computational complexity

ex05_complexity.py[a].
To run this code example, you must install the **texttable** component from https://github.com/foutaise/texttable

[a] ProTip: This is a clickable link

Overcalculation in recursive Fibonacci



Demo

Discussion

How can overcalculation be eliminated?

Memoization example

ex06_complexity_optimized.py[a].
To run this code example, you must install the **texttable** component from https://github.com/foutaise/texttable

[a] ProTip: This is a clickable link

Efficiency of a function

What is function efficiency?

The resources the function uses, usually measured in either the time required to complete or memory used.

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

Complexity

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

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 runtime 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 the number of operations they make
- **NB!** You cannot restrict input size when analyzing for the best case (e.g., $n = 1$ is not best case, it's just one particular input size)

Run time complexity

- Runtime 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

Upper bound on algorithm complexity

First, we introduce an asymptotic upper bound on algorithm complexity. This is the most useful, as it tells us that the algorithm cannot have worse complexity than its upper bound.

Example: The bubble sort algorithm

- Its worst case happens when trying to sort a list that is already sorted, but in reverse order
- Each comparison results to the compared elements switching positions
- It also leads to the longest runtime when tested with actual data

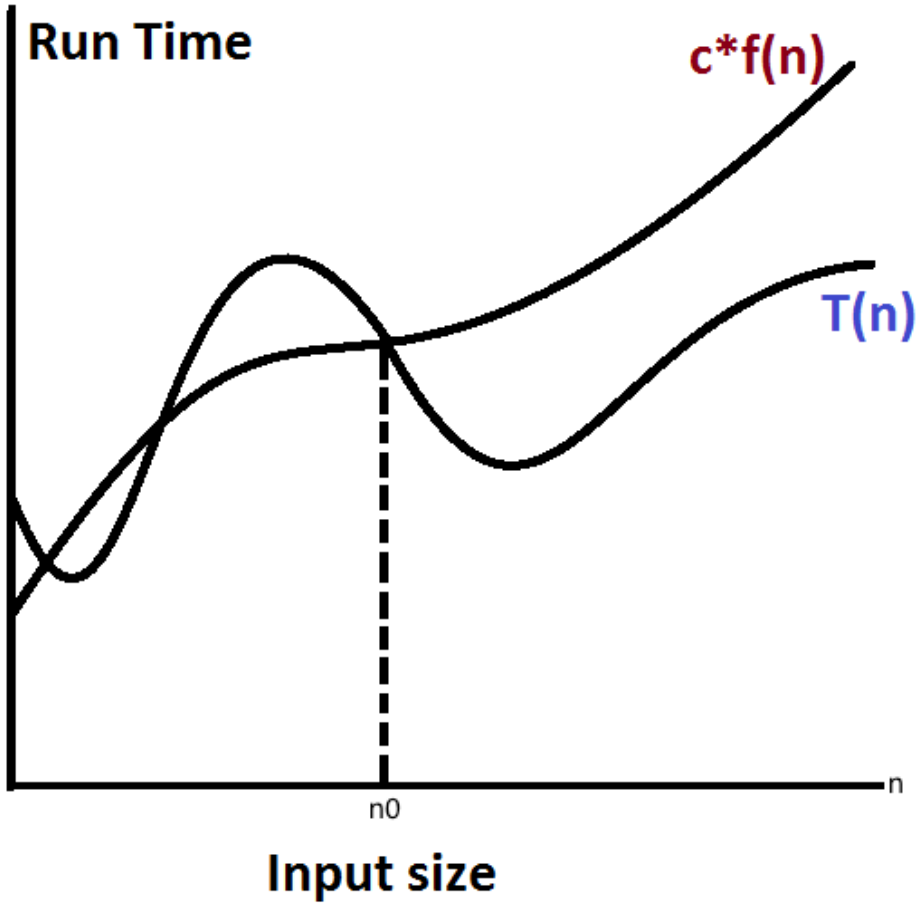
"Big-O" notation

- Consider a function $f : \mathbb{N} \to \mathbb{R}$
- Consider a function $T : \mathbb{N} \to \mathbb{N}$ that provides the number of operations performed by the algorithm; T's input represents the size of the algorithm's input data, and its output is the number of operations carried out.
- For example, a linear search in a list will compare each element with the one being search. Therefore, if the search traverses the entire list of n elements, then $T(n) = n$.

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 $T(n) = 13 * n^3 + 42 * n^2 + 2 * n * \log_2 n + 3 * \sqrt{n}$, and $f(n) = n^3$, 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 then go to 0.
- Next, we also introduce a lower bound on complexity

"Big-omega" notation

Lower bound on algorithm complexity

Next, we introduce an asymptotic lower bound on algorithm complexity. This is not that useful by itself, but it will become so in the next step. The lower bound tells us that the algorithm cannot have better complexity than this.

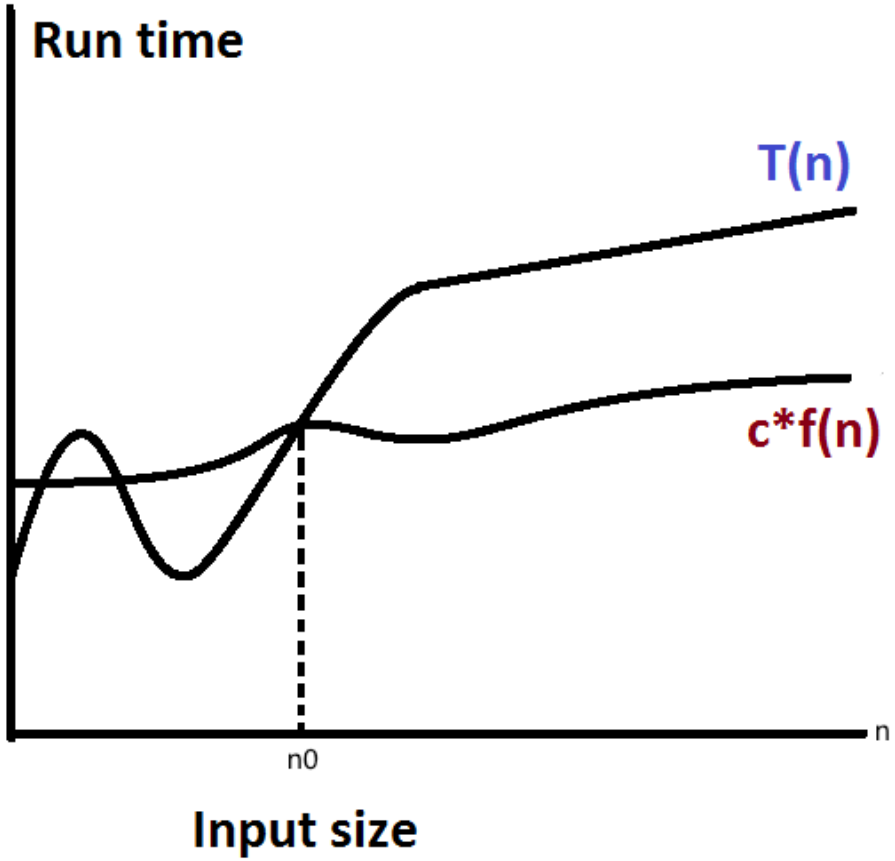
Example: The bubble sort algorithm

- Its best case happens when trying to sort a list that is already sorted (and in the correct order!)
- No elements are being switched, and the algorithm finishes after going through the list a single time
- It leads to the shortest runtime when tested with actual data

"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 $T(n) = 13 * n^3 + 42 * n^2 + 2 * n * \log_2 n + 3 * \sqrt{n}$ and $f(n) = n^3$, 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 establishing a lower bound on an algorithm's complexity.

"Big-theta" notation

Tight bound on algorithm complexity

Finally, we introduce a tight bound on algorithm complexity. This is the most useful **when it exists**. This tells us the exact computational complexity of the algorithm.

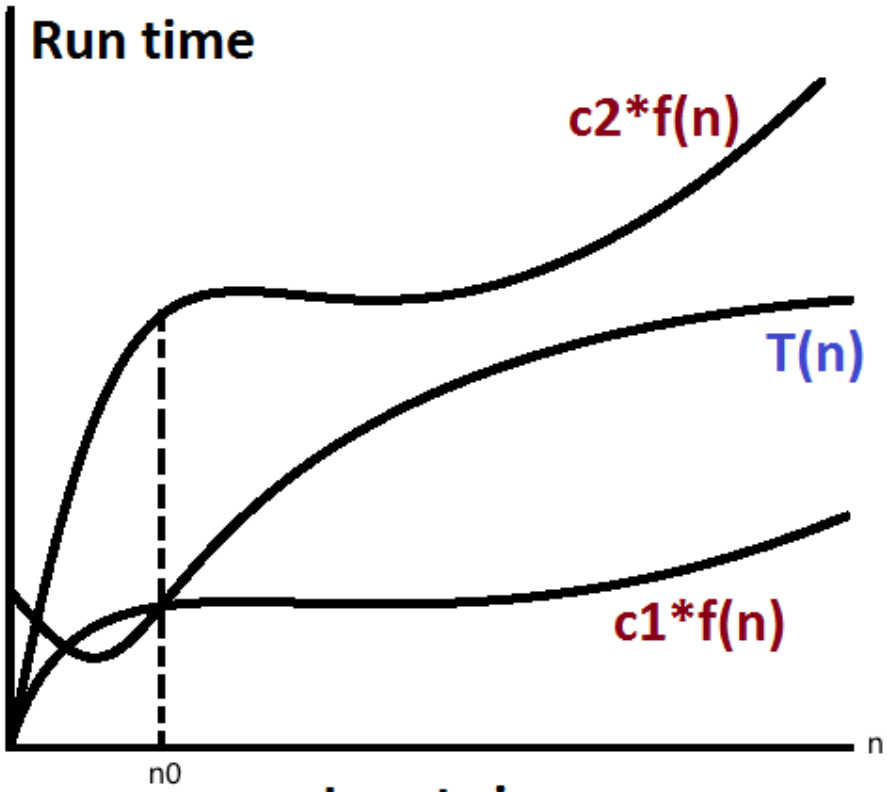
Example: The bubble sort algorithm

- Its best case is $O(n)$ (going through the list a single time, we visit each element once)
- Its worst case is $O(n^2)$ (we need $(n-1)$ comparisons to move the first element to the last position, and so on)
- In this case the upper and lower bounds are different

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

```
1    for i in data_list:
2        # do something 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

```
1    def f(n : int) -> int:
2        s = 0
3        for i in range(1, n + 1):
4            s = s + 1
5        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

```
1    def f(n : int) -> None:
2        i = 0
3        while i <= n:
4            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

```
1    def f(data : list) -> bool:
2        '''
3        Return True if list contains an even number
4        '''
5        poz = 0
6        while poz < len(data) and data[poz] % 2 !=0:
7            poz += 1
8        return poz < len(data)
```

- 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

```
1    def f(data : list) -> bool:
2        poz = 0
3        while poz < len(data) and data[poz] % 2 !=0:
4            poz += 1
5        return poz < len(data)
```

- To make things easier, let's assume the searched value is in the list, with a uniform probability for it to be on any position of the list
- AC - the **while** can be executed $1, 2, ...n$ times, with the same probability. 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

```
1    def f(n : int) -> None:
2        for i in range(1, 2 * n - 1):
3            for j in range(i + 2, 2 * n + 1):
4                # constant time operation
5                s = i + j
```

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

Summation - examples

```
1     def f(n : int) -> None:
2         for i in range(1, 2 * n - 1):
3             j = i + 1
4             cond = True
5             while j < 2 * n and cond:
6                 if someCondition: # constant time
7                     cond = False
```

- Best Case - line 8 executes on the first iteration each time,

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

- 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

```
1    def f(n : int) -> None:
2        for i in range(1, 2 * n - 1):
3            j = i + 1
4            cond = True
5            while j < 2 * n and cond:
6                if someCondition: # constant time
7                    cond = False
```

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

Big-O cheat sheet

`https://www.bigocheatsheet.com/`

Recurrences

What is a recurrence?

A recurrence is a mathematical formula defined recursively.

Example I - Node count of complete 3-ary tree

- 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:
$$\begin{cases} N(h) = 1, h = 0 \\ N(h) = 3 * N(h-1) + 1, h \geq 1 \end{cases}$$
- The number of nodes of a complete 3-ary tree of height 0 is 1 (just the root).
- A complete 3-ary tree of height $h, h \geq 1$ 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: $N(h) = 1 + 3^1 + 3^2 + ... + 3^h = \sum_{i=0}^{h} 3^i$.

Example II - Recursive list summation

```
1    def recursiveSum(data : list) -> int:
2        '''
3        Compute the sum of the numbers in a list
4        '''
5        if data == []: # base case
6            return 0
7        # recursion step
8        return data[0] + recursiveSum(data[1:])
```

- n represents the length of the list
- The recurrence is: $T(n) = \begin{cases} 1, n = 0 \\ T(n-1) + 1, n > 0 \end{cases}$
- **NB!** Function must first check whether it was called with an empty list

Example II - Recursive list summation

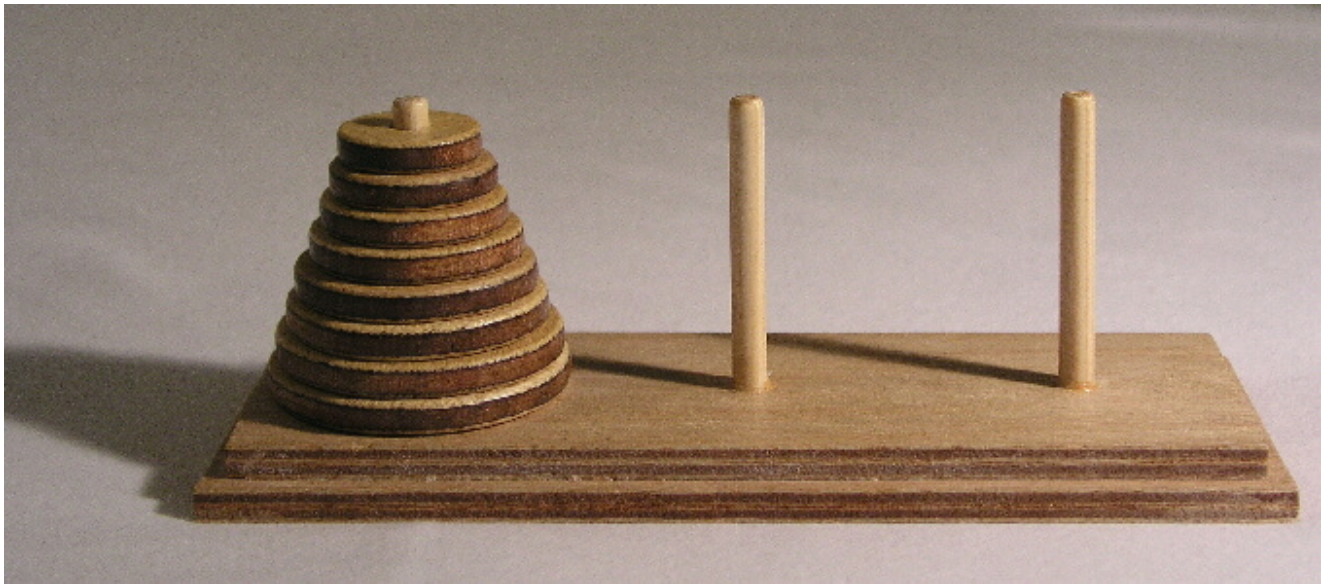
- Solving the reccurence:

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

- $T(n) = T(n-1) + 1$
- $T(n-1) = T(n-2) + 1$
- $T(n-2) = T(n-3) + 1 \Rightarrow T(n) = n + 1 \in \Theta(n)$

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

Start with three rods and a number of discs of successively increasing radii 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

```
1   def hanoi(n : int, x : str, y : str, z : str) -> None:
2       '''
3       n - number of disks on the x stick
4       x - name of source stick
5       y - name of destination stick
6       z - name of intermediate stick
7       '''
8       if n==1:
9           print("disk 1 from ", x, " to ", y)
10          return
11      hanoi(n - 1, x, z, y)
12      print("disk ", n, " from ",x ," to ", y)
13      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:
  $$\mathrm{T(n)} = \begin{cases} 1, n = 1 \\ 2T(n-1) + 1, n > 1 \end{cases}$$
- $T(n) = 2T(n-1) + 1,\ T(n-1) = 2T(n-2) + 1,$
  $T(n-2) = 2T(n-3) + 1,\ldots,\ T(1) = T(0) + 1$
- $T(n) = 2T(n-1) + 1,\ 2T(n-1) = 2^2T(n-2) + 2,$
  $2^2T(n-2) = 2^3T(n-3) + 2^2,\ldots,2^{n-2}T(2) = 2^{n-1}T(1) + 2^{n-2}$
- 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

Tower of Hanoi

ex07_hanoi.py[a].
To run this code example, you must install the **texttable** component from https://github.com/foutaise/texttable

[a] ProTip: This is a clickable link

Memory space complexity

What is the memory 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 Big-O, "Omega" and "Theta" notation.

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

Example I - List summation

```
1   def iterative_sum(data : list) -> int:
2       '''
3       Compute the sum of the numbers in a list
4       data - input list
5       return the sum of the numbers
6       '''
7       res = 0
8       for nr in data:
9           rez += nr
10      return rez
```

- We do not allocate additional memory based on the length of the initial list, so $T(n) = 1 \in \Theta(1)$.

Example I - List summation

```
1   def recursive_sum(data : list) -> int:
2       '''
3       Compute the sum of the numbers in a list
4       data - input list
5       return the sum of the numbers
6       '''
7       if data == []: # base case
8           return 0
9       # recursion step
10      return data[0] + recursive_sum(data[1:])
```

- The recurrence is: $T(n) = \begin{cases} 0, n = 1 \\ T(n-1) + n - 1, n > 1 \end{cases}$
- **NB!** Function must first check whether it was called with an empty list

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