Data Strucutres and Algorithms

Nithin

July 2, 2023

Table of Contents

What is Algorithm?

As per Donald Knuth

Algorithm

A definite, effective and finite process that receives input and produces an output

- Definite : steps are clear, concise and unambigious
- Effective : you can perform each operation precisely
- Finite : finite number of steps

Analysis

When two programs solve the same problem, Analysis is finding answer to the question which one is better?

Better on?

- Readability :

Better on?

- Readability : changes with programming language

Better on?

- Readability : changes with programming language
- Number of Lines :

Better on?

- Readability : changes with programming language
- Number of Lines : changes with programming language

Better on?

- Readability : changes with programming language
- Number of Lines : changes with programming language
- Amount of computing resources :

Better on?

- Readability : changes with programming language
- Number of Lines : changes with programming language
- Amount of computing resources : changes with programming language

Better on?

- Readability : changes with programming language
- Number of Lines : changes with programming language
- Amount of computing resources : changes with programming language
- Run time :

Better on?

- Readability : changes with programming language
- Number of Lines : changes with programming language
- Amount of computing resources : changes with programming language
- Run time : changes with processor speed, compiler and programminglanguage

An example : Checking the run time

our first example

Big-O Notation

Requirement

To charactrize an algorithm's efficiency in terms of execution time, independet of any particular program or computer

Solution

To quanitfy the algorithm in terms of number of operations or steps

$T(n)$

$T(n)$ is a function that indicates the time an algorithm takes to solve a problem of size n

Example 1

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

- For sum_of_n, we can take the basic compute step as the assignment operations
- In sum_of_n following are the assignment operations
  - sum $= 0$
  - sum$+= n$
- $T(n) = n + 1$
- We are only interested in the dominant term in $T(n)$, beacuse as n increases faster compared to other terms, i.e it overpowers the rest

Big-O

The dominant term in $T(n)$, which can be termed as order of magnitude function. Big-O $\implies$ Biggest Order frametitleCommon Big-O functions

Quiz 1

What is the Big-O for the program given below :

```
a=5
b=6
c = 10
for i in range(n):
    for j in range(n):
        x=i*i
        y=j*j
        z=i*j
for k in range(n):
    w = a * k + 45
    v=b*b
d = 33
```

Quiz 1

What is the Big-O for the program given below :

```
a=5
b=6
c = 10
for i in range(n):
    for j in range(n):
        x=i*i
        y=j*j
        z=i*j
for k in range(n):
    w = a * k + 45
    v=b*b
d = 33
```

$$T(n) = 3n^2 + 2n + 4 \implies O(n^2)$$

Anagram

A

string is an anagram of another if second is simply a rearrangment of the first. For example python and typhon

Solution 1: Checking off

```
        def anagram_sol1(word1, word2):
            word2_list = list(word2)
            index1=0
            is_anagram = True
            while index1 < len(word1) and is_anagram:
                index2=0
                is_continue = True
                while index2 < len(word2_list) and
    is_continue:
                    if word1[index1] == word2_list[index2]:
                        word2_list[index2] = None
                        is_anagram = True
                        is_continue = False
                    else:
                        is_anagram = False
                        is_continue = True
                        index2+=1
                index1+=1
            return is_anagram
```

Solution 1: Big-O

Example

Each letter in word1 has to iterate a maximum of n locations to find a match. That is

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

Solution 2: Sort and Compare

```python
def anagram_sol2(word1, word2):
    word1_list = list(word1)
    word2_list = list(word2)
    word1_list.sort()
    word2_list.sort()
    index=0
    while index < len(word1_list):
        if word1_list[index] != word2_list[index]:
            return False
        index+=1
    return True
```

Solution 2: Big-O

There is a single iteration of n if there is a match but the sort operation takes the precedence due to it's $O(n^2)$ or $O(n \log n)$ complexity

Solution 3: Brute Force

This tries to exhaust all possibilities. Here we genrate all possible anagrams of word1 and matches this with word2. For a word of length n, there are $n!$

Solution 4: Count and Compare

```
def anagram_sol4(word1,word2):
    counter1 = [0]*26
    counter2 = [0]*26
    offset = ord('a')
    for letter in word1:
        counter1[ord(letter)-offset]+=1
    for letter in word2:
        counter2[ord(letter)-offset]+=1
    for index in range(26):
        if counter1[index] != counter2[index]:
            return False
    return True
```

Solution 4: Count and Compare

```python
    def anagram_sol4(word1,word2):
        counter1 = [0]*26
        counter2 = [0]*26
        offset = ord('a')
        for letter in word1:
            counter1[ord(letter)-offset]+=1
        for letter in word2:
            counter2[ord(letter)-offset]+=1
        for index in range(26):
            if counter1[index] != counter2[index]:
                return False
        return True

```

$$T(n) = 2n + 26 \implies O(n)$$

The solution above can run in linear time but required more space requirements than the other solutions

Linear Strucutres

What is Linear Data Strucutre?

- Data structures in which each element stays its position relative to the elements before and after.
- Examples are Stacks, Queues, Deques and Lists
- The difference are in the way items are added or removed

Stack

What is a Stack?

A stack is an ordered collection of items where the addition of new item and the removal of existing items always takes place at the same end

- also known as Last-In-First-Out(LIFO)
- Newer items are in the top and older items are at the bottom
- Browser Back button is an example of stack

Stack-ADT: Python Implementation

- **Stack()** : creates new stack. Needs no parameters rather returns an empty stack
- **push(item)** :adds a new item to top of the stack. It needs the item and returns nothing
- **pop()** : removes the top item from the stack. It needs no parameter and return the item. stack is modified
- **peek()** : returns top item from the stack. No item parameter is required and did not modify the stack
- **is_empty()** : tests to see whether stack is empty. It needs no parameter and returns a boolean value
- **size()** : returns the number of items in the stack. It needs no parameter and returns an integer value

Stack ADT: Python Implementation

```python
class Stack:
    def __init__(self) -> None:
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        return self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[-1]

    def size(self):
        return len(self.items)
```

Stack Example: Paranthesis Checker

```
def para_check(symbol_str):
    # creating symbol templates and the key - value pairs
    open_symbol = {'(':0, '[':1, '{':2}
    close_symbol = {')':0, ']':1, '}':2}
    # initialising the stack
    para_stack = Stack()
    #to check the unbalance condition during symbol iteration and exit the loop is exists
    is_check = True
    index = 0
    while index < len(symbol_str) and is_check:
        symbol = symbol_str[index]
        #checking for opening symbols
        if symbol in open_symbol.keys():
            para_stack.push(symbol)

```

```
            #checking for closing symbols
            elif symbol in close_symbol.keys():
                # check is stack is empty and still a close symbol exists resulting in unbalance
                if para_stack.is_empty():
                    # exit the loop if exists
                    is_check = False
                # check if the closing symbol is equal to opening symbol , if not, exit the loop
                elif close_symbol[symbol] != open_symbol[para_stack.peek()] :
                    is_check = False
                else:
                    # if both of the above condition fails, then there is a balance exists and pop out the symbol
                    para_stack.pop()

```

```python
                else:
                    pass
                index+=1
            # to check if the stack is empty and check the balance flag
            if para_stack.is_empty() and is_check:
                return f"balanced"
            else:
                return f"unbalanced"


```