

Data Structures and Algorithms

Nithin

July 2, 2023

Table of Contents

1 Algorithm Analysis

- Big-O
- Anagram Example

2 Basic Data Structures

- Linear Data Structure
- Stack

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 unambiguous

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 programming language

An example : Checking the run time

our first example

Big-O Notation

Requirement

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

Solution

To quantify 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

```
1  def sum_of_n(n):  
2      total = 0  
3      for i in range(n):  
4          total+=i  
5      return total  
6  
7
```

- 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 $T(n)$, because 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

Common Big-O functions

Quiz 1

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

```
1  a=5
2  b=6
3  c = 10
4  for i in range(n):
5      for j in range(n):
6          x=i*i
7          y=j*j
8          z=i*j
9  for k in range(n):
10     w = a * k + 45
11     v=b*b
12 d = 33
13
```

Quiz 1

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

```
1  a=5
2  b=6
3  c = 10
4  for i in range(n):
5      for j in range(n):
6          x=i*i
7          y=j*j
8          z=i*j
9  for k in range(n):
10     w = a * k + 45
11     v=b*b
12 d = 33
13
```

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

Anagram

A

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

Solution 1: Checking off

```
1  def anagram_sol1(word1, word2):
2      word2_list = list(word2)
3      index1=0
4      is_anagram = True
5      while index1 < len(word1) and is_anagram:
6          index2=0
7          is_continue = True
8          while index2 < len(word2_list) and
is_continue:
9              if word1[index1] == word2_list[index2]:
10                 word2_list[index2] = None
11                 is_anagram = True
12                 is_continue = False
13             else:
14                 is_anagram = False
15                 is_continue = True
16                 index2+=1
17             index1+=1
18         return is_anagram
19
```

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

```
20 def anagram_sol2(word1, word2):
21     word1_list = list(word1)
22     word2_list = list(word2)
23     word1_list.sort()
24     word2_list.sort()
25     index=0
26     while index < len(word1_list):
27         if word1_list[index] != word2_list[index]:
28             return False
29         index+=1
30     return True
31
```

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 generate all possible anagrams of **word1** and matches this with **word2**. For a word of length n , there are $n!$

Solution 4: Count and Compare

```
1  def anagram_sol4(word1, word2):
2      counter1 = [0]*26
3      counter2 = [0]*26
4      offset = ord('a')
5      for letter in word1:
6          counter1[ord(letter)-offset]+=1
7      for letter in word2:
8          counter2[ord(letter)-offset]+=1
9      for index in range(26):
10         if counter1[index] != counter2[index]:
11             return False
12     return True
13
```

Solution 4: Count and Compare

```
1  def anagram_sol4(word1, word2):
2      counter1 = [0]*26
3      counter2 = [0]*26
4      offset = ord('a')
5      for letter in word1:
6          counter1[ord(letter)-offset]+=1
7      for letter in word2:
8          counter2[ord(letter)-offset]+=1
9      for index in range(26):
10         if counter1[index] != counter2[index]:
11             return False
12     return True
13
```

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

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

What is Linear Data Structure?

- 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

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 Example: Paranthesis Checker

```
1  def para_check(symbol_str):
2      # creating symbol templates and the key - value
   pairs
3      open_symbol = {'(':0, '[':1, '{':2}
4      close_symbol = {')':0, ']':1, '}':2}
5      # initialising the stack
6      para_stack = Stack()
7      #to check the unbalance condition during symbol
   iteration and exit the loop is exists
8      is_check = True
9      index = 0
10     while index < len(symbol_str) and is_check:
11         symbol = symbol_str[index]
12         #checking for opening symbols
13         if symbol in open_symbol.keys():
14             para_stack.push(symbol)
15
```

```

1         #checking for closing symbols
2         elif symbol in close_symbol.keys():
3             # check is stack is empty and still a close
symbol exists resulting in unbalance
4             if para_stack.is_empty():
5                 # exit the loop if exists
6                 is_check = False
7                 # check if the closing symbol is equal to
opening symbol , if not, exit the loop
8                 elif close_symbol[symbol] != open_symbol[
para_stack.peek()] :
9                     is_check = False
10                else:
11                    # if both of the above condition fails,
then there is a balance exists and pop out the symbol
12                    para_stack.pop()
13

```

```
1         else:
2             pass
3             index+=1
4             # to check if the stack is empty and check the
balance flag
5             if para_stack.is_empty() and is_check:
6                 return f"balanced"
7             else:
8                 return f"unbalanced"
```