IIT Madras
BSc Degree

Home Week-1 Week-3

PDSA - Week 2

Complexity

The **complexity** of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of input data. There are two main complexity measures of the efficiency of an algorithm:

Space complexity

The space complexity of an algorithm is the amount of memory it needs to run to completion.

Generally, space needed by an algorithm is the sum of the following two components:

- Fixed part(C) - Size of code
- Variable Part(S_x) - Depend on input size, to store in memory

  Total Space $(T(x)) = C + S_x$

Time complexity

The time complexity of an algorithm is the amount of computer time it needs to run to completion. Count the number of operations executed by the processor.

Time complexity calculated in three types of cases:

- Best case
- Average case
- Worst Case

Growth rate of functions

The number of operations for an algorithm is usually expressed as a function of the input.

For Example:

```
s = 0 #1
For i in range(n): #n+1
    for j in range(n): #n(n+1)
        s = s + 1 #n^2
print(s)#1
```

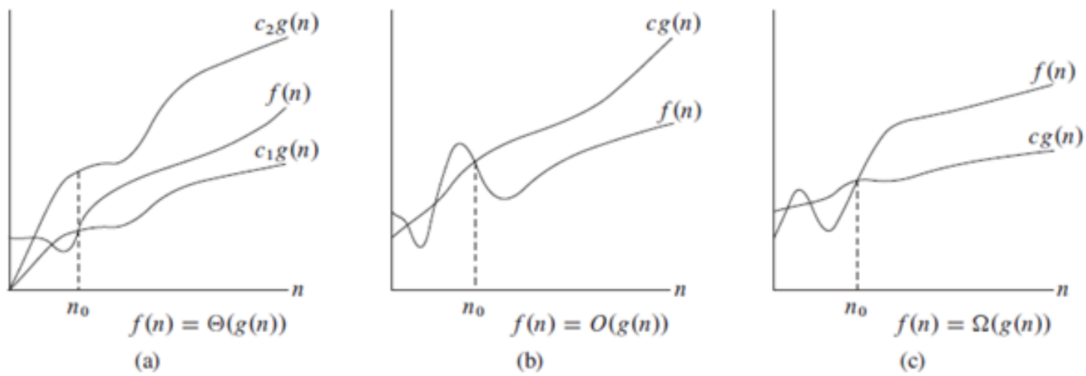
Function for given code is :

$f(n) = 2n^2 + 2n + 3$

Ignore all the constant and coefficient just look at the highest order term in relation to n. So $f(n)$ is proportional to n^2

Notations to represent complexity

- Big-Oh(O) - Upper bound
- Omega(Ω) - Lower bound
- Theta(Θ) - Tightly bound



source = https://www.dotnetlovers.com/images/coolnikhilj2256c883d1-b9fc-46e9-b225-588ac5063c3d.png

Calculate complexity

```
a = 10
b = 20
s = a + b
print(s)
```

Complexity ? O(1)

Complexity for single loop

```
s = 0
For i in range(n):
    s = s + 1
```

Complexity ? O(n)

Complexity for nested two loop

```
s = 0
For i in range(n):
    for j in range(n):
        s = s + 1
```

Complexity ? $O(n^2)$

Complexity for nested three loop

```
s = 0
For i in range(n):
    for j in range(n):
        for k in range(n)
            s = s + 1
```

Complexity ? $O(n^3)$

Complexity for combination of all

```
s = 0
For i in range(n):
    s = s + 1
s = 0
For i in range(n):
    for j in range(n):
        s = s + 1
s = 0
For i in range(n):
    for j in range(n):
        for k in range(n)
            s = s + 1
```

Complexity ? $O(n^3)$

Complexity for recursive solution

```
def factorial(n)
    if (n == 0):
        return 1
    return n * factorial(n - 1)
```

Recurrence relation ? T(n) = T(n-1) + O(1) = 1+1+1...n times

Complexity ? $O(n)$

Complexity for recursive solution

```
def merge(A,B):
    #statement block for merging two sorted array
def mergesort(A):
    n = len(A)
    if n <= 1:
        return(A)
    L = mergesort(A[:n//2])
    R = mergesort(A[n//2:])
    B = merge(L,R)
    return(B)
```

Recurrence relation ? T(n) = 2T(n/2)+ O(n)

Complexity ? $O(nlogn)$

Searching Algorithm

Linear search and Binary search working

https://www.cs.usfca.edu/~galles/visualization/Search.html

Implementation of Linear Search or Naïve Search

```
def naivesearch(v,l):
  for x in l:
    if v == x:
      return(True)
  return(False)
```

Analysis

Best Case - $O(1)$

Average Case - $O(n)$

Worst Case - $O(n)$

Implementation of Binary Search

```python
def binarysearch(L,v):
    if L == []:
        return(False)
    mid = len(L)//2
    if v == L[mid]:
        return(True)
    if v < L[mid]:
        return(binarysearch(L[:mid],v))
    else:
        return(binarysearch(L[mid+1:],v))
```

***Due to use of slicing, this implementation takes O(n) time, We can implement binary search without slicing in following way:-**

Iterative Implementation

```python
def binarysearch(L, v):
    s = 0
    e = len(L) - 1
    m = 0
    while s <= e:
        m = s + (e - s) // 2
        if L[m] < v:
            s = m + 1
        elif L[m] > v:
            e = m - 1
        else:
            return m
    return -1
```

Recursive Implementation

```python
def binarysearch(L,v,s,e):
    if e - s == 0:
        return(v==L[s])
    mid = (e + s)//2
    if v == L[mid]:
        return(True)
    if v < L[mid]:
        return(binarysearch(L,v,s,mid-1))
    else:
        return(binarysearch(L,v,mid+1,e))
```

Analysis

Best Case - $O(1)$

Average Case - $O(logn)$

Worst Case - $O(logn)$

Sorting Algorithm

Selection Sort

Working

7 VISUALGO.NET / en /sorting

BUB SEL INS MER QUI R-Q COU

RADIX SORT

LOGIN

We use cookies to improve our website.

By clicking ACCEPT, you agree to our use of Google Analytics for analysing user behaviour and improving user experience as described in our Privacy Policy.

By clicking reject, only cookies necessary for site functions will be used.

Source - https://visualgo.net/en/sorting

Implementation

```python
def selectionsort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        mpos = i
        for j in range(i+1,n):
            if L[j] < L[mpos]:
                mpos = j
        (L[i],L[mpos]) = (L[mpos],L[i])
    return(L)
```

Analysis

Best Case - $n + (n-1) + (n-2)...2 + 1 = n(n+1)/2 = O(n^2)$

Average Case - $n + (n-1) + (n-2)...2 + 1 = n(n+1)/2 = O(n^2)$

Worst Case - $n + (n-1) + (n-2)...2 + 1 = n(n+1)/2 = O(n^2)$

Stable - No

Sort in Place - Yes

Insertion Sort

Working

7 VISUALGO.NET / en / sorting

BUBBLE SORT SEL INS MER QUI R-Q COU RAD

LOGIN

Source - https://visualgo.net/en/sorting

Implementation

```python
def insertionsort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        j = i
        while(j > 0 and L[j] < L[j-1]):
            (L[j],L[j-1]) = (L[j-1],L[j])
            j = j-1
    return(L)
```

Analysis

Best Case - $1 + 1 + 1...1 + 1(ntimes) = n = O(n)$

Average Case - $n+(n-1)+(n-2)...2+1 = n(n+1)/2 = O(n^2)$

Worst Case - $n+(n-1)+(n-2)...2+1 = n(n+1)/2 = O(n^2)$

Stable - Yes

Sort in Place - Yes

Merge Sort

Working

7 VISUALGO.NET / en /sorting
BUB SEL INS MER QUICK SORT R-Q COU
RAD
LOGIN

We use cookies to improve our website.
By clicking ACCEPT, you agree to our use
of Google Analytics for analysing user
behaviour and improving user experience
as described in our Privacy Policy.
By clicking reject, only cookies necessary
for site functions will be used.

Source - https://visualgo.net/en/sorting

Implementation

```
def merge(A,B):
    (m,n) = (len(A),len(B))
```

```
    (C,i,j) = ([],0,0)
    while i < m and j < n:
        if A[i] <= B[j]:
            C.append(A[i])
            i += 1
        else:
            C.append(B[j])
            j += 1
    while i < m:
        C.append(A[i])
        i += 1
    while j < n:
        C.append(B[j])
        j += 1
    return C


def mergesort(A):
    n = len(A)
    if n <= 1:
        return(A)
    L = mergesort(A[:n//2])
    R = mergesort(A[n//2:])
    B = merge(L,R)
    return(B)
```

Analysis

Best Case - $n + n + n...logn\ times = nlogn = O(nlogn)$

Average Case - $n + n + n...logn\ times = nlogn = O(nlogn)$

Worst Case - $n + n + n...logn\ times = nlogn = O(nlogn)$

Stable - Yes

Sort in Place - No

Complexity of python data structure's method

Keep in mind before using for efficiency.

https://wiki.python.org/moin/TimeComplexity

List methods

Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
Append[1]	O(1)	O(1)
Pop last	O(1)	O(1)
Pop intermediate[2]	O(n)	O(n)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)
Del Slice	O(n)	O(n)
Set Slice	O(k+n)	O(k+n)
Extend[1]	O(k)	O(k)
Sort	O(n log n)	O(n log n)
Multiply	O(nk)	O(nk)
x in s	O(n)	
min(s), max(s)	O(n)	
Get Length	O(1)	O(1)

Dictionary methods

Operation	Average Case	Amortized Worst Case
k in d	O(1)	O(n)
Copy[3]	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item[1]	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration[3]	O(n)	O(n)

Set Methods

Operation	Average case	Worst Case
x in s	O(1)	O(n)
Union s|t	O(len(s)+len(t))	
Intersection s&t	O(min(len(s), len(t))	O(len(s) * len(t))
Multiple intersection s1&s2&..&sn		(n-1)*O(l) where l is max(len(s1),..,len(sn))
Difference s-t	O(len(s))	
s.difference_update(t)	O(len(t))	
Symmetric Difference s^t	O(len(s))	O(len(s) * len(t))
s.symmetric_difference_update(t)	O(len(t))	O(len(t) * len(s))