VINOD KUMAR VERMA, PGT(CS), KV OEF KANPUR &
SACHIN BHARDWAJ, PGT(CS), KV NO.1 TEZPUR

PROGRAM EFFICIENCY

Measuring the performance of CODE

Introduction

- Efficiency refers to the quality of Algorithm i.e. code must perform the task in minimum execution time and resources.
- It is important to measure the efficiency of algorithm before applying them on large scale i.e. on bulk of data.
- Nowadays most of application are online where prompt response is required, if efficiency of algorithm is not checked then the site will crash and organization may loose their customer/business because of slow speed.

Introduction

- The performance of Algorithm depends upon
  - INTERNAL FACTORS
    - Time Required to Run
    - Space(or Memory) required to Run
  - EXTERNAL FACTORS
    - Size of input to the algorithm
    - Speed of the computer on which it is run
    - Quality of the compiler.
- ***Since the external factors are controllable to some extent, our major focus is to handle the internal factors.***

Computational Complexity

- Computation involves problems to be solved and algorithm to solve them
- Complexity involves study of how much resource like TIME and SPACE is needed to run the algorithm
- Effectiveness means that the algorithm carries out its intended function correctly
- Efficiency means algorithm should be correct with the best possible performance

Estimating Complexity of Algorithms

- In Previous Slide we already understood that 2 major factors responsible for efficiency of algorithm are TIME TAKEN and AMOUNT OF SPACE.
- Out of two, TIME TAKEN is more important factor to consider.
- TIME COMPLEXITY of a program(for given input) is the number of elementary instruction that this program executes. This number is computed with respect to the size n of the input data.

Big-O Notation

- The Big-O notation is used to depict an algorithm's growth rate. The growth rate determines the algorithm's performance when its input size grows.
- Through Big-O, the upper bound of an algorithm's performance is specified i.e. if algorithm takes O(n^2) time; this means algorithm will take at the most n^2 steps for input size n.
- ***O(n) means algorithm will take n steps for input n***
- ***O(1) means algorithm will take 1 step to perform action for input n***

Big-O Notation

SIZE / COMPLEXITY	10	20	40	100	400
n^2	100	400	1600	10000	160000
2^n	1024	1048576	10^{12}	1.26×10^{30}	Very Big…

Performance of algorithm is inversely proportional to the wall clock time it records for a given input size. Program with a bigger O run slower than program with a smaller O

Dominant term

- It is the term which affect the most on algorithm's performance.
- For example if the term is : $ax^2 + bx + c$ (for constant a, b, c), then we can see the maximum impact on the algorithm's performance will be of the term ax^2. So only dominant term is included with Big-O notation. If the algorithm's has performance is $O(n^2)$ then for larger n, the n^2 dominates.

Common Growth Rate

TIME COMPLEXITY		EXAMPLE
O(1)	Constant	Push in Stack
O(log N)	log	Finding entry in sorted array
O(N)	linear	Finding entry in unsorted array
O(N log N)	n log n	Sorting n items by divide and conquer
$O(N^2)$	quadratic	Shortest path between two nodes in a graph
$O(N^3)$	cubic	Simultaneous linear equation
$O(2^n)$	exponential	The Tower of Hanoi problem

Guidelines for computing complexity

- Select the computational resource you want to measure. Normally we have to measure time complexity.
- Look out for the variable which makes algorithm work more or less. It may be single or multiple variables. This will be our size of input.
- After this try to see, if there are different cases inside it, such as when algorithm gives best performance, when gives worst performance and when algorithm takes between the two cases.

Calculating Complexity

- Five guidelines for finding out the time complexity of a piece of code are :
  - Loops
  - Nested Loops
  - Consecutive statements
  - If-then-else statements
  - Logarithmic complexity

Loop

- The running time of loop is equal to the running time of statements inside the loop multiplied by number of iteration. For example:

```
for i in range(n):
        a = a + 2
```

Loop executes n times

All the steps in this loop takes constant time, C

So, total time taken = C * n = Cn , here n is a dominant term, So efficiency is O(n)

Nested Loop

- To compute complexity of nested loop, analyze inside out. For nested loops, total running time is the product of the sizes of loops:

```
for i in range(n):
    for j in range(n):
        S = S + 1
```

Outer loop n times

Inner loop n times

All steps takes constant time C

So total time taken = n * n * c = cn^2 i.e. $O(n^2)$

Consecutive Statements

- To compute complexity of consecutive statements, simply add the time complexities of each statement

```
a = a + 1                  # Constant time C1
for i in range(n):         # Loop1 n times
        s = s + 1          # Constant time C2
for j in range(n):         # Outer loop n times
        for k in range(n): # Inner loop n times
        x = x + 1          # Constant time C3
```

So total time taken = C1 + n * C2 + C3*n^2 i.e. O (n^2)

VINOD KUMAR VERMA, PGT(CS), KV OEF KANPUR &
SACHIN BHARDWAJ, PGT(CS), KV NO.1 TEZPUR

If-then-else statements

- To compute time complexity of if-then-else, we consider the worst case running time i.e. time taken by the test, plus time taken by either then part of the else part, whichever is larger.

```
if len(list1)!=len(list2):          <----- Constant time C1
    return false                    <----- Constant time C2
else:
    for i in range(n):              <----- Loop executes n times
        if list1[i]!=list2[i]:      <----- Constant time C3
            return false            <----- Constant time C4
```

So, total time taken = C1 + C2 + (C3+C4) * n i.e. O(n), ONLY DOMINANT TERM IS USED.

Logarithmic Complexity

- Means that an algorithms performance time has logarithmic factor e.g. an algorithm O(log N) if it takes constant time to cut the program size by fraction (usually by ½) i.e. after every iteration the number of possibility to repeat the loop is reduced by 1/2 like in BINARY SEARCHING.

Best , Average and Worst Case Complexity

- Best Case means an algorithm is performing its intended operation using minimum number of steps.
- Worst Case means an algorithm is performing its intended operation using maximum number of steps.
- Average case means between the Best and Worst Case i.e. average number of steps
- TAKE AN EXAMPLE: with Linear Searching:
- If item we are searching is at 1st place then it will be its BEST CASE, if item to search it at last place in list or not in list then it will be its WORST CASE for any other position it will be its AVERAGE CASE.

Determining the complexity of a program that checks if a number n is prime

- First Approach -

```
n = int(input("Enter any number :"))
isprime = True
for i in range(2,n):
        if n % i == 0:
                isprime=False
                break
if isprime:
        print("Number is Prime")
else:
        print("Number is not Prime")
```

Line	Cost
n = int(input("Enter any number :"))	Constant Time C1
isprime = True	Constant Time C2
for i in range(2,n):	Loop repeats n times
if n % i == 0:	Constant time C3
isprime=False / break	Constant time C4
if isprime:	Constant time C5
print("Number is Prime")	Constant time C6
print("Number is not Prime")	Constant time C7

- **So total time taken will be : C1+C2+(C3+C4)*n+C5+C6+C7 i.e O(n), considering the dominant term which is n**

Determining the complexity of a program that checks if a number n is prime

- Second approach $\sqrt{n}$ *approach*

```
n = int(input("Enter any number :"))
isprime = True
i=2
while i*i <=n:
        if n % i == 0:
                isprime=False
                break
        i=i+1
if isprime:
        print("Number is Prime")
else:
        print("Number is not Prime")
```

Statement	Time
n = int(input("Enter any number :"))	Constant Time C1
isprime = True	Constant Time C2
i=2	Constant time C3
while i*i <=n:	Loop repeats $\sqrt{n}$ times
if n % i == 0:	Constant time C4
isprime=False	Constant time C5
i=i+1	Constant time C6
if isprime:	Constant time C7

- **So total time taken: O($\sqrt{n}$) because $\sqrt{n}$ is the dominant term. Hence Second approach for prime test is better than first approach because $\sqrt{n} < n$**

Determining the complexity of a program that searches for an element in array

- Option 1 : Linear Search

```
def linearSearch(mylist,item):
        n = len(mylist)
        for i in range(n):
                if item == mylist[i]:
                        return i
        return None
```

Constant Time C1

Loop repeats n times

Constant time C2

Constant time C3

Constant time C4

So total time taken : C1 + (C2+C3) * n + C4 i.e. O(n)

Only dominant term is taken

Determining the complexity of a program that searches for an element in array

- Option 2 : Binary Search

```
def binarySearch(mylist,item):
    low = 0                          ← C1
    high = len(mylist)-1             ← C2
    while Low<=high:                 ← C3
        mid = int((low+high)/2)      ← C4
        if item ==  mylist[mid]:     ← C5
            return mid               ← C6
        elif item<mylist[mid]:       ← C7
            high = mid-1             ← C8
        else:
            low = mid +1             ← C9
    return None                      ← C10
```

Loop repeats max $\log_2 N$ times because every time segment becomes half in size

So total time taken : C1+C2+log2N(C3+C4+…C10) i.e. $O(\log_2 N)$

Only dominant term is taken

How many times above while loop executes?

- How many times can you divide N by 2 until you have 1? This is because in binary searching, search begins with N and reduces to its half after every iteration and stops when the search segment size reduce to 1 element. So if loop repeats k times then in formula this would be:

$$N/2^k = 1$$

$$2^k = N$$

Taking log2 on both sides:

$$\log_2(2^k) = \log_2 N$$

$$k * \log_2(2) = \log_2 N$$

$$k * 1 = \log_2 N$$

$$k = \log_2 N$$

This means you can divide log N times until you have everything divided this means loops repeats at max log N times

Comparing Option 1 and Option 2 we can say that Option 2 is better as $\log_2 N < n$