A Review on Time Complexities at Different Optimization Levels

Anjali Dhariwal
PhD Scholar, Faculty of Computer Applications, Jodhpur National University, India
anjalidhariwal1181@gmail.com

***Abstract*—General study has shown that time complexity of an algorithm is always evaluated at the source code and is mostly determined by worst-case time complexity represented by Big-O notation. Whereas, my analysis says time complexity of an algorithm differs after applying code optimization techniques. Hence, it is also quite possible that an algorithm that runs faster on a machine happens to be slower on the other. My research work would basically draw a relative study of time complexities at different levels, Source Code, Machine Independent Code, Machine Dependent Code.**

Keywords– Code Optimisation Techniques, Time Complexity, Big-O Notation.

I. INTRODUCTION

TO solve any problem, we need to follow a finite number of steps which in computer terminology we call Algorithm. It requires a great effort to design a correct and optimized algorithm else it may lead to erroneous results and overuse of resources. Once the algorithm is designed, we need to analyse the time and space taken by it and check if there are any bottlenecks to be removed. Mostly, the space complexity remains constant and can be optimised with proper data structure. The time complexity entirely depends on how the instructions are being computed and executed. Time complexity can be greatly reduced at different levels of optimizations. While analysing an algorithm we often consider Best, Average and Worst case of runtime. Best case is not really of concern due it its triviality and is practically meaningless. Mostly, it makes more sense to do worst case analysis, identifying the inputs that take the longest time. Worst Case Time Complexity gives an upper bound for any input, assuring algorithm cannot take longer than this. Evaluating average case time complexity can be very difficult and is often not very apparent. Therefore, in most algorithms we typically consider Worst Case Time Complexity only. Worst case time complexity that gives an upper bound for an algorithm is represented by Big-O notation as given below.

Given functions f(n) and g(n), we say that f(n) is of O(g(n)) if and only if f(n) <= c g(n) for n>=n0 where c and n0 are constants, f(n) and g(n) are functions over non-negative integers. In my research I will consider algorithms of following time complexities:

- logarithmic: O(log n)
- linear O(n)
- quadratic O(n2)
- polynomial O(nk), k 1
- exponential O(an), n> 1

As a common practice, time complexity of an algorithm or a program is computed on the source code. But my observation reveals that there may be a difference in the time complexity when calculated on the source code and when calculated after various optimizations applied on the algorithm. Time complexity if reduced even by a constant may show significant difference on a large set of data. We shall calculate worst-case time complexities on an algorithm at different levels, Source Code, Machine Independent Code and Machine Dependent Code for the same set of data. And it is also possible that an algorithm that runs faster on a machine turns out to be slower on the other depending on the processor configuration. So it is inappropriate to determine the time complexity of an algorithm at the source code level. We shall therefore draw a relative study between all three to identify at which level should the time complexity be calculated for generic algorithms.

II. CODE OPTIMISATION TECHNIQUES AND METHOD

In order to evaluate time complexities of an algorithm at different levels of optimisations, we need to understand different code optimisation techniques and method to find time complexity of a given algorithm.

1. Code Optimisation Techniques

Optimization [1] is the most desirable goals in software engineering and is much important as it reduces runtime resources and at times memory space also. Code optimization can be done at three levels, High Level Optimization, Intermediate Code Optimization and Low Level Optimization. Following are techniques for optimization at three different levels -

- High Level Optimization –
  - Inlining – Replacing function call with function body.
  - Partial Evaluation – It includes Tail Recursion Elimination, Loop Reordering, Array Alignment, Padding and evaluation of those components which can be evaluated.
- Intermediate Code Optimization –
  - Common Subexpression Elimination
  - Constant Propagation
  - Jump Threading
  - Loop-invariant Code Motion

- o Dead Code Elimination
- o Strength Reduction
- • Low Level Optimization –
- o Register Allocation
- o Instruction Scheduling for Pipelined Machines
- o Loop Unrolling
- o Instruction Reordering
- o Delay Slot Filling
- o Utilizing Features of Specialized Components, e.g. floating point.
- o Branch Prediction

Some optimizations involve trade-offs, e.g. more memory for faster execution and should also be cost-effective, i.e., benefits of optimization must be worth the effort.

Compiler Code Optimizer [1] are designed such that they sit between the front end and the code generator and works basically with the intermediate code. Compiler code optimization provides the following –

- • Inlining Small Functions
- • Code Hoisting
- • Dead Store Elimination
- • Eliminating Common Sub-Expressions
- • Loop Unrolling
- • Loop Optimization: Code motion, Induction variable elimination, and Reduction in strength.

Inlining Small Functions – By repeatedly inserting the function code instead of calling it, saves the calling overhead and enable further optimizations, while inlining large functions will make the executable too large.

Code Hoisting – Moving computation outside the loop saves computing time.

Dead Store Elimination - Compiler may safely ignore many of the operations that compute the values of variables that are never used.

Loop Unrolling – Since the loop exit checks cost CPU time, if it is known how many times a loop is repeated, it can be unrolled.

Code Motion – any formula inside the loop that always computes the same value can be moved before the loop.

2. Complexity of an Algorithm

To analyse an algorithm is to determine the resources (such as time and storage) necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length/size. Usually, the complexity of an algorithm is a function relating the input length/size to the number of fundamental steps (time complexity) or fundamental storage locations (space complexity). The fundamental steps and storage locations are, of course, dependent on the physics of the underlying computation machinery.

Time complexity estimates depend on what we define to be a fundamental step. For the analysis to correspond usefully to the actual execution time, the time required to perform a fundamental step must be guaranteed to be bounded above by a constant. One must be careful here; for instance, some analyses count an addition of two numbers as one step. This assumption will be false in many contexts. For example, if the numbers involved in a computation may be arbitrarily large, the time required by a single addition can no longer be assumed to be constant.

Space complexity estimates depend on what we define to be a fundamental storage location. Such storage must offer reading and writing functions as fundamental steps. Most computers offer interesting relations between time and space complexity. For example, on a Turing machine the number of spaces on the tape that play a role in the computation cannot exceed the number of steps taken. In general, space complexity is bounded by time complexity. Many algorithms that require a large time can be implemented using small space.

Example:-

1.	READ A	1
2.	READ B	1
3.	Z = A + B	1
4.	PRINT Z	1
5.	END	

Complexity = 1 + 1 + 1 + 1
= 4
= O(1)

1.	READ N	1
2.	SUM = 0	1
3.	FOR I = 1 TO N DO	N+1
a.	READ X	N
b.	SUM = SUM + X	N
4.	AVG = SUM / N	1
5.	PRINT AVG, SUM	1
6.	END	

Complexity= 1+1+N+1+N+N+1+1
= 5 + 3N
= 3N
= O(N)

7.	READ N	1
8.	FOR I = 1 TO N DO	
	N+1	
9.	READ X(I)	
	N	
10.	FOR I = 1 TO N DO	N+1
11.	FOR J = I TO N DO	N(N-1)/2) - 1
12.	IF X (J) < X(I)	N(N-1)/2
13.	THEN SWAP (X (I), X (J))	
14.	ENDFOR	
15.	ENDFOR	
16.	FOR I = 1 TO N DO	N+1
17.	WRITE X (I)	N
18.	ENDFOR	

19. END

Complexity=1+N+1+N+1+(N(N-1)/2)-1+(N(N-1)/2)+N+1+N

= N2 + 4N + 3 ≈ N2

= O(N2)

When analysing algorithms[5], we are often interested in analysing the best, average, and worst cases of running time. Typically, best case performance is not really of concern due to triviality. Algorithms may be modified to make best case performance trivial for small input by hardcoding. In these cases, the best case performance is effectively meaningless! Often, it is of more concern to perform worst case analysis, i.e. identifying the inputs that cause the algorithm to have the longest running time (or use the most space) and identifying the running time and usage bounds.

Why is this useful?

- The worst case running time of an algorithm gives an upper bound on the running time for any input. Knowing this provides a guarantee that the algorithm never takes any longer.
- For some algorithms, the worst case may occur fairly often.
- Often, the "average case" is roughly as bad as the worst case.

Finally, in some cases we may be interested in the average case running time of an algorithm, where we would use probabilistic analysis to examine particular algorithms. This doesn't surface too much, as what constitutes an "average input" is often not apparent. Often, we would then assume that all inputs of a particular size are equally likely (uniform distribution). This assumption is often violated in practice, so we might modify an algorithm to be randomized, to enable a probabilistic analysis and expected running time. An algorithm may run faster on certain data sets than on others. Finding the average case can be very difficult, so typically algorithms are measured by the worst-case time complexity. Also, in certain application domains (e.g., air traffic control, surgery) knowing the worst-case time.

Examples of Time-Complexity –

Best Case: Array is empty or element is found at first index. Easy to compute but irrelevant.

Worst Case: Array has to be searched up till the end. Easy to compute, provides upper bound for the running.

Average Case: Depends on inputs and statistical distribution of the inputs. Difficult to compute.

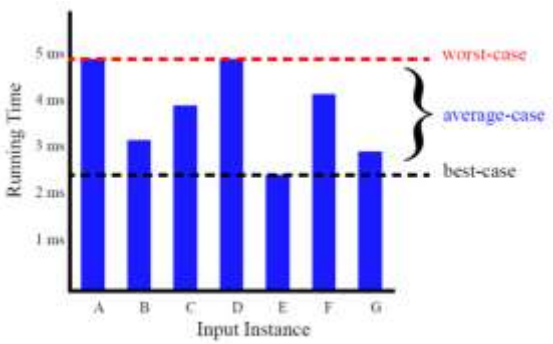

Fig 1: Time Complexity- Best Case, Worst Case and Average Case

Informally, an algorithm can be said to exhibit a growth rate on the order of a mathematical function if beyond a certain input size n, the function f(n) times a positive constant provides an upper bound or limit for the run-time of that algorithm. In other words, for a given input size n greater than some n0 and a constant c, an algorithm can run no slower than c × f(n). This concept is frequently expressed using Big O notation [2].

For example, since the run-time of insertion sort grows quadratically as its input size increases, insertion sort can be said to be of order $O(n^2)$.

Big O notation is a convenient way to express the worst-case scenario for a given algorithm, although it can also be used to express the average-case— for example, the worst-case scenario for quicksort is $O(n^2)$, but the average case run-time is O(n lg n).

Note: Average-case analysis is much more difficult than worst-case analysis.

Given functions f(n) and g(n), we say that f(n) is of O(g(n)) if and only if

f(n) <= c g(n) for n>=n0 where c and n0 are constants, f(n) and g(n) are functions over non-negative integers.

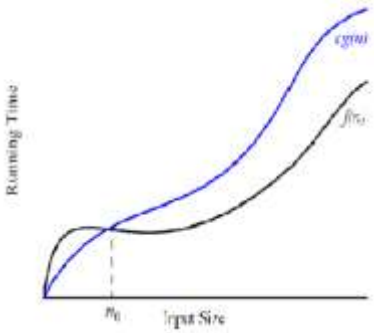

Fig 2: Representation of Big-O Notation

Just as Big O describes the upper bound, we use Big Omega to describe the lower bound. Big Theta describes the case where the upper and lower bounds of a function are on

the same order of magnitude [3].

Table1: Orders of Growth[3]

Notation	Common name	Limit test (note limit may not exist)
$f(n) \in O(g(n))$	Asymptotic upper bound	$\lim \frac{f(x)}{g(x)} < \infty$
$f(n) \in o(g(n))$	Asymptotically negligible	$\lim \frac{f(x)}{g(x)} = 0$
$f(n) \in \Omega(g(n))$	Asymptotic lower bound	$\lim \frac{f(x)}{g(x)} > 0$
$f(n) \in \omega(g(n))$	Asymptotically dominant	$\lim \frac{f(x)}{g(x)} = \infty$
$f(n) \in \Theta(g(n))$	Asymptotically tight bound	$0 < \lim \frac{f(x)}{g(x)} < \infty$

III. Graphical Representation of Notations [3]

O-notation (Upper Bound)

This notation gives and upper bound for a function to within a constant factor. We write f(n)=Og(n) if there are positive constants n_o and c such that to the right of n_o. the value of f(n) always lies on or below cg(n).

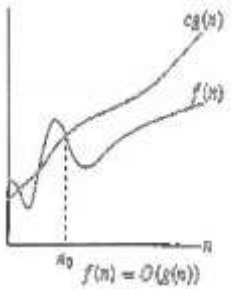

Fig 3: Representation of O-Notation

Ω-notation (Upper Bound)

This notation gives and lower bound for a function to within a constant factor. We write f(n)= Ωg(n) if there are positive constants n_o and c such that to the right of n_o. the value of f(n) always lies on or abov cg(n).

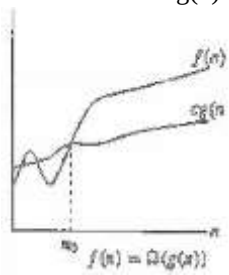

Fig 4: Representation of Ω-Notation

θ-notation (same order)

This notation gives and lower bound for a function to within a constant factor. We say f(n)= θg(n) if there exist positive constants n_o, c_1 and c_2 such that to the right of n_o the value of f(n) always lies between $c_1g(n)$ and $c_2g(n)$ inlusive.

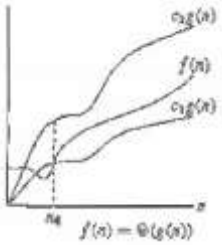

Fig 5: Representation of Θ-Notation

IV. Comparison of different time complexities [4]

Different Time-Complexities are

- logarithmic: O(log n)
- linear O(n)
- quadratic O(n2)
- polynomial O(nk), k 1
- exponential O(an), n> 1

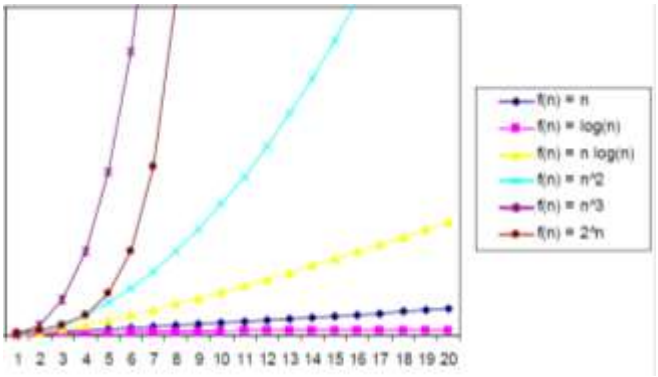

Fig 6: Graphical representation of different time complexities

Let's compare some basic time complexities.

EXAMPLES

a) Constant time — O(1).

```
1. def constant(n):
2.     result = n * n
3.     return result
```

There is always a fixed number of operations.

b) Logarithmic time — O(log n).

```
1 def logarithmic(n):
2     result = 0
3     while n > 1:
4         n = n/2
5         result += 1
6     return result
```

The value of n is halved on each iteration of the loop. If n = 2x then log n = x. How long would the program below take to execute, depending on the input data?

c) Linear time — O(n).

```
1 def linear(n, A):
2     for i in xrange(n):
3         if A[i] == 0:
4             return 0
5     return 1
```

Let's note that if the first value of array A is 0 then the program will end immediately. But remember, when analyzing time complexity we should check for worst cases. The program will take the longest time to execute if array A does not contain any 0.

d) Quadratic time — O(n2).

```
1 def quadratic(n):
2     result = 0
3     for i in xrange(n):
4         for j in xrange(i, n):
5             result += 1
6     return result
```

The result of the function equals 1/2 • (n • (n + 1)) = 1/2 • n2 + 1/2 • n (the explanation is in the exercises). When calculating the complexity we are interested in a term that grows fastest, so we not only omit constants, but also other terms (1/2 • n in this case). Thus we get quadratic time complexity. Sometimes the complexity depends on more variables (see example below).

e) Linear time — O(n + m).

```
1 def linear2(n, m):
2     result = 0
3     for i in xrange(n):
4         result += i
5     for j in xrange(m):
6         result += j
7     return result 2
```

f) Exponential and Factorial time

It is worth knowing that there are other types of time complexity such as factorial time O(n!) and exponential time O(2n). Algorithms with such complexities can solve problems only for very small values of n, because they would take too long to execute for large values of n.

An increase in speed of computer does not necessarily does not increases by the data size by same amount which can be solved by the system. Table below shows five different algorithms A1….A5 having different time complexities. Considering 1 unit time then if A1 can solve / compute 100 data then within same unit time algorithm A5 can solve only 6.5 data. Now if we increase the speed of computer by a factor of 10 then under normal assumptions we state that the data handling increases to 10 times which is not a fact as depicted in table. Algorithm A1 can handle 10 times but algorithm A5 can handle only 1.5 times of previous data size. Thus we can state that nearly increasing the speed of computer, the data size cannot be increased proportionally. Hence, the complexity of algorithm is deciding factor to handle large data within given unit time. From the above table we can conclude that algorithm A1 as best time complexity whereas A5 has worst time complexity. Thus, during the development of algorithm 1 must try to keep the complexity of the algorithm in neighbourhood of 'N' such that large data size can be solved if the system speed is increased.

Table2: Increase in Runtime of an algorithm is not proportional to the increase in Data Size

Algorithm	Complexity	1 Unit	10s	Difference	Priority	1 Min	1Hour
A1	N	100	1000	10	1	6×10^{7}	3.6×10^{8}
A2	N^2	10	31.6	3.16	3	4896	2.0×10^{5}
A3	N^3	3.4	10	2.9	4	244	1897
A4	$N\log_2 N$	22	144	6.1	2	39	153
A5	2^N	6.5	10	1.5	5	15	21

V. RELATED WORK

There has been work to optimize the code to reduce time complexity of an algorithm but there has been no comparative study given between the time complexities at source code and after optimisation.

REFERENCES

[1] Optimization Techniques by Sekar April 21, 2009.
[2] Fundamentals of the Analysis of Algorithm Efficiency by Manjunath, May 29, 2010.
[3] Complexity & Algorithm Analysis by J Paul Gibson, 2012. Link- http://www -public.it -sudparis.eu/~gibson/Teaching/MAT7003/
[4] From Codility, Chapter-3, Time Complexity
[5] CIS 121 – Data Structures and Algorithms with Java, Spring 2017