

# CMPSC-122: Intermediate Programming

Spring 2018

Department of Computer Science & Engineering  
The Pennsylvania State University

---

## 1. Analysis of algorithms

When you write a program you should be concerned about the resource needs of the program. The two main resources to consider are *time* and *memory*. These are separate resources and depending on the situation, you may end up choosing an algorithm that uses more of one resource in order to use less of the other. Understanding this will allow you to produce better code. The resource to optimize for depends on the application and the computing system. Does the program need to finish execution within a restricted amount of time? Does the system have a limited amount of memory? There may not be one correct choice. It is important to understand the pros and cons of each algorithm and data structure for the application at hand.

The amount of resources consumed often depends on the amount of data you have. Intuitively, it makes sense that if you have more data you will need more space to store the data. It will also take more time for an algorithm to run. Algorithm analysis does not answer the question “How much of a resource is consumed to process  $n$  pieces of data?”. What we really care about is the growth rate of resource consumption with respect to the data size. Generally, we perform the following types of analysis:

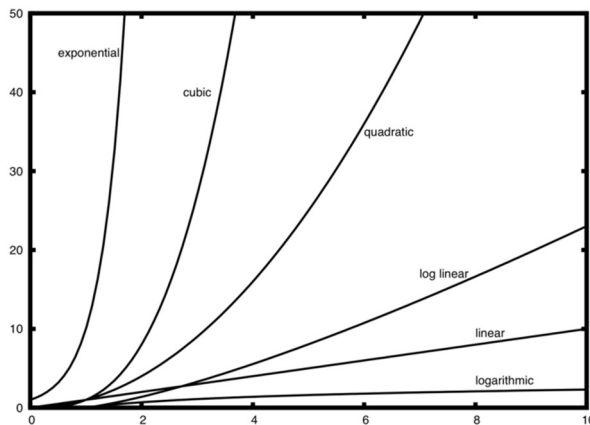
- Worst-case – The maximum number of steps taken on any instance of size  $n$ .
- Best-case – The minimum number of steps taken on any instance of size  $n$ .
- Average case – An average number of steps taken on any instance of size  $n$ .

### 1.1 Growth rates

Algorithm analysis is all about understanding growth rates. That is as the amount of data gets bigger, how much more resource will the algorithm require? Typically, we describe the resource growth rate of a piece of code in terms of a function. Some of the different growth rates from most efficient to least efficient are:

- Constant Growth Rate: A constant resource need is one where the resource need does not grow. That is processing one piece of data takes the same amount of resource as processing 1 million pieces of data.
- Logarithmic Growth Rate: A logarithmic growth rate is a growth rate where the resource needs grows by one unit each time the data is doubled. This effectively means that as the amount of data gets bigger, the curve describing the growth rate gets flatter (closer to horizontal but never reaching it).

- **Linear Growth Rate:** A linear growth rate is a growth rate where the resource needs and the amount of data is directly proportional to each other. That is the growth rate can be described as a straight line that is not horizontal.
- **Log Linear Growth Rate:** A log linear growth rate is a slightly curved line. The curve is more pronounced for lower values than higher ones
- **Quadratic Growth Rate:** A quadratic growth rate is one that can be described by a parabola.
- **Cubic Growth Rate:** While this may look very similar to the quadratic curve, it grows significantly faster
- **Exponential Growth Rate:** An exponential growth rate is one where each extra unit of data requires a doubling of resource.



$f(n)$	Name
1	Constant
$\log n$	Logarithmic
$n$	Linear
$n \log n$	Log Linear
$n^2$	Quadratic
$n^3$	Cubic
$2^n$	Exponential

Notice that when  $n$  is small, the functions are not very well defined with respect to one another. It is hard to tell which is dominant. However, as  $n$  grows, there is a definite relationship and it is easy to see how they compare with one another.

## 1.2 Asymptotic Analysis

The asymptotic behavior of a function  $f(n)$  refers to the growth of  $f(n)$  as  $n$  gets large. We typically ignore small values of  $n$ , since we are usually interested in estimating how slow the program will be on large inputs. In asymptotic analysis, we evaluate the performance of an algorithm in terms of input size (we do not measure the actual running time) and calculate how does the time (or space) taken by the algorithm increases with the input size.

Let us consider the problem of searching an element in a sorted list of  $n$  elements. One way to search is linear search (order of growth is linear) and other way is binary search (order of growth is logarithmic). Based on their order of growth, after certain value of  $n$ , the binary search will definitely start taking less time compared to the linear search (even if we run binary search on a slow machine).

### 1.2.1 Solving recurrences

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Recurrences are generally used in divide-and-conquer paradigm. Let us consider  $T(n)$  to be the running time on a problem of size  $n$ . If the problem size is small enough, say  $n < c$  where  $c$  is a constant,

the straightforward solution takes constant time, which is written as  $O(1)$ . A recurrence relation can be solved using the following methods:

- Substitution Method: we guess a bound and using mathematical induction we prove that our assumption was correct.
- Recursion Tree Method: a recurrence tree is formed where each node represents the cost.
- Master's Theorem: the most used technique to find the complexity of a recurrence relation.

### 1.3 Big-O Notation

The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that does not depend on machine specific constants. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The Big-O notation is one of the formal notational methods for stating the growth of resource needs (efficiency and storage) of an algorithm and it defines the upper bound of an algorithm (worst case).

$$T(n) \text{ is } O(f(n)) \text{ iff for some constants } c \text{ and } n_0, T(n) \leq c f(n) \text{ for all } n \geq n_0$$

This basically means that  $f(n)$  describes the upper bound for  $T(n)$ . One of the simplest ways to think about Big-O notation is that it is basically a way to apply a rating system for your algorithms. It tells you the kind of resource needs you can expect the algorithm to exhibit as your data gets bigger and bigger. From best (least resource requirements) to worst, the rankings are:  $O(1)$ ,  $O(\log n)$ ,  $O(n)$ ,  $O(n \log n)$ ,  $O(n^2)$ ,  $O(n^3)$ ,  $O(2^n)$ . One important advantage of Big-O notation is that it makes algorithms much easier to analyze, since we can conveniently ignore low-order terms. For example, an algorithm that runs in time:

$$10n^3 + 24n^2 + 3n \log n + 144$$

is still a cubic algorithm, since:

$$10n^3 + 24n^2 + 3n \log n + 144 \leq 10n^3 + 24n^3 + 3n^3 + 144n^3 \leq (10 + 24 + 3 + 144)n^3 = O(n^3)$$

### 1.4 Examples

Let's start with the simple piece of Python code shown below. Although this program does not really do anything, it is instructive to see how we can take actual code and analyze performance.

```

a=5
b=10
c=15
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 = 56

```

assignment operations = 3

loop performed  $n$  times

loop performed  $n$  times

assignment statements = 3

the three statements are performed  $n^2$  times =  $3n^2$

loop performed  $n$  times



assignment statements = 2

the two statements are performed  $n$  times =  $2n$

assignment operations = 1

This gives us  $T(n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4$ . By looking at the exponents, we can easily see that the  $n^2$  term will be dominant and therefore this fragment of code is  $O(n^2)$ . The other terms as well as the coefficient on the dominant term can be ignored as  $n$  grows larger.



Now, let's consider the following implementation of linear search:

```
def linear_search(alist, x):  
    for i in range(len(alist)):  In a list of size  $n$ , loop could be performed  
        if alist[i] == x:  up to  $n$  times (worse case)  
            return i  
    return -1
```

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For linear search, the worst case happens when the element to be searched ( $x$  in the above code) is not present in the list. When  $x$  is not present, the `linear_search()` function compares it with all the elements of `alist[]` one by one. Therefore, the worst case time complexity of linear search would be  $O(n)$ .

Finally, let's consider the two implementations of binary search:

*#Iterative implementation*

```
def binarySearch(alist, item):  
    first = 0  
    last = len(alist)-1  
    found = False  
  
    while first<=last and not found:  
        midpoint = (first + last)//2  each comparison eliminates about half of the  
        if alist[midpoint] == item:  remaining items from consideration  
            found = True  
        else:  
            if item < alist[midpoint]:  
                last = midpoint-1  
            else:  
                first = midpoint+1  
  
    return found
```


The code within this loop is constant, meaning that no matter what size is one iteration of the loop, it will take the same number of operations (more or less) and thus, the question really becomes how many times will this loop run?

The only non-constant part of the time cost in this code is the number of times the while loop runs. Each step of the while loop cuts the range in half, until our range has just one element left. So the question is, how many times can we split a list of size  $n$  until we get down to one element? If we start with  $n$  elements, about  $\frac{n}{2}$  elements will be left after the first comparison. After the second comparison, there will be about  $\frac{n}{4}$ . Then  $\frac{n}{8}$ ,  $\frac{n}{16}$ , and so on:

$$n \times \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} \times \dots = 1 \quad \text{how many times we can split?} \quad n \times (1/2)^y = 1$$

Solving for  $y$ :

$$\begin{aligned} n \times (1/2)^y &= 1 \\ n \times 1^y / 2^y &= 1 \\ n \times (1/2^y) &= 1 \\ n / 2^y &= 1 \\ n &= 2^y \\ \log_2 n &= \log_2 2^y \\ \log_2 n &= y \end{aligned}$$

 The number of times we must divide  $n$  in half to get down to 1 is  $\log_2 n$ . Thus, the total time cost is  $O(\log n)$

```

# Recursive implementation
def binarySearch (arr, l, r, x):
    if r >= l:
        mid = l + (r - l) / 2  → constant time

        if arr[mid] == x:
            return mid  → constant time

        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x) → same problem of size (n/2)

        else:
            return binarySearch(arr, mid+1, r, x) → same problem of size (n/2)

    else:
        return -1

```

The recurrence relation of this binary search implementation can be written as:

$$T(n) = T(n/2) + c$$

Using the master theorem (see Appendix 1):

$a = 1, b=2, f(n) = c$  thus  $c = 0$

$\log_b a ? c$

$\log_2 1 ? 0$

$0 = 0$

Following from the second case of the master theorem  $T(n) = O(n^c \log n) = O(\log n)$

## Appendix 1. The Master Theorem

In the analysis of algorithms, the master theorem provides a cookbook solution in asymptotic terms (using Big-O notation) for recurrence relations of types that occur in the analysis of many divide and conquer algorithms. The Master theorem allows us to easily calculate the running time of recursive algorithm of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \text{where } a \geq 1, b > 1$$

where:

- $n$  is the size of the problem
- $a$  is the number of subproblems in the recursion
- $n/b$  is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- $f(n)$  is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems

Now, let  $a$  be an integer greater than or equal to 1 and  $b$  be a real number greater than 1. Let  $c$  be a positive real number and  $d$  a nonnegative real number. Given a recurrence of the form:

$$T(n) = \begin{cases} aT(n/b) + n^c & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases}$$

Then:

- ✓ if  $\log_b a < c$ ,  $T(n) = O(n \log_b a)$
- ✓ if  $\log_b a = c$ ,  $T(n) = O(n^c \log n)$
- ✓ if  $\log_b a > c$ ,  $T(n) = O(n^{\log_b a})$

Example:

$$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

$$a = 8, b=2, f(n) = 1000n^2 \text{ thus } c = 2$$

$$\log_b a \text{ ? } c$$

$$\log_2 8 \text{ ? } 2$$

$$3 > 2$$

Following from the third case of the master theorem  $T(n) = O(n^{\log_b a}) = O(n^3)$

## Appendix 2. Big-O efficiency of basic list operations

Operation	Big-O Efficiency
index []	$O(1)$
index assignment	$O(1)$
append	$O(1)$
pop()	$O(1)$
pop(i)	$O(n)$
insert(i,item)	$O(n)$
del operator	$O(n)$
iteration	$O(n)$
contains (in)	$O(n)$
get slice [x:y]	$O(k)$
del slice	$O(n)$
set slice	$O(n+k)$
reverse	$O(n)$
concatenate	$O(k)$
sort	$O(n \log n)$
multiply	$O(nk)$

### Appendix 3. Big-O efficiency of basic dictionary operations

Operation	Big-O Efficiency
copy	$O(n)$
get item	$O(1)$
set item	$O(1)$
delete item	$O(1)$
contains (in)	$O(1)$
iteration	$O(n)$



## Appendix 4. Big-O Cheat Sheet

### Data structures

Data Structure	Average cases			Worst cases		
	Insert	Delete	Search	Insert	Delete	Search
Array/stack/queue	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Linked list	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly linked list	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Hash table	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Binary search tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$

### Graphs

Node/edge management	Storage size	Add vertex	Add edge	Remove vertex	Remove edge	Query
Adjacency list	$O( V  +  E )$	$O(1)$	$O(1)$	$O( V  +  E )$	$O( E )$	$O( V )$
Adjacency matrix	$O( V ^2)$	$O( V ^2)$	$O(1)$	$O( V ^2)$	$O(1)$	$O(1)$

### Sorting algorithms

Algorithm (applied to an array)	Time complexity		
	Best cases	Average cases	Worst cases
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$
Quick Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$

### Searching algorithms

Algorithm	Data structure	Worst case
Sequential search	Array and linked list	$O(n)$
Binary search	Sorted array and binary search tree	$O(\log(n))$
Depth-first search (DFS)	Graph of $ V $ vertices and $ E $ edges	$O( V  +  E )$
Breadth-first search (BFS)	Graph of $ V $ vertices and $ E $ edges	$O( V  +  E )$

## References

<http://aofa.cs.princeton.edu/10analysis/>

[https://www.tutorialspoint.com/design\\_and\\_analysis\\_of\\_algorithms/analysis\\_of\\_algorithms.htm](https://www.tutorialspoint.com/design_and_analysis_of_algorithms/analysis_of_algorithms.htm)

<https://cathyatseneca.gitbooks.io/data-structures-and-algorithms/analysis/>

[https://www.packtpub.com/sites/default/files/downloads/4874OS\\_Appendix\\_Big\\_O\\_Cheat\\_Sheet.pdf](https://www.packtpub.com/sites/default/files/downloads/4874OS_Appendix_Big_O_Cheat_Sheet.pdf)