Data Structures & Algorithms

Week 3

Searching, Sorting and Complexity Analysis

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 2

- Data structures can be classified as being either linear or non-linear.
- Data structure where data elements are arranged sequentially or linearly where each and every element is attached to its previous and next adjacent is called a linear data structure.
- Arrays, stacks, queues, and linked lists are examples of linear data structures.
- Data structures where data elements are not arranged sequentially or linearly are called non-linear data structures.
- Trees and graphs are examples of non-linear data structures.
- The hash table is a special data structure that can be classified as either linear or non-linear.

Content

- Efficiency of algorithms
- Complexity analysis
- Search algorithms
- Sort algorithms

Part 1

Efficiency of algorithms

1.1 Introduction

- Algorithms were introduced in lesson 1; they were defined and a few examples discussed.
- How do we determine how effective an algorithm is? Well, the obvious answer would be that it solves the problem that it was intended to.
- The next question to ask is whether this is the only way to measure the efficiency of an algorithm? In lesson 1 (see slide 42) it was stated *inter alia*, ".. the best measure to use is to express the running time of the algorithm as a function of the input size n (this is what is used in all literature regarding this topic), and compare these different functions corresponding to running times. This then is a truly objective measure…"
- In that lesson we stated that we would revisit the issue of evaluation of efficiency of algorithms in a later lesson; well here we are. This is it. This lesson is about using different tools to measure the efficiency of algorithms with a view to understanding how they perform.

1.2 Runtime

- One way to measure the cost of an algorithm is to measure how long it takes to execute some code; that is, to measure it's actual running time using the computer's clock.
- The process is also referred to as benchmarking or profiling. The process is not complicated. The user will collect several datasets of the same size and run them; the average time taken to execute these datasets is recorded. Then several datasets of a different size are run in a similar fashion; the average is computed in a similar manner. By performing this operation with different datasets a fair estimation can be made of the running time of any dataset without actually running it.
- Let us demonstrate this using a simple Python program. We write a program that will simply count from 1 to a given number. Thus, the problem size is the number. You start with the number 10,000,000, time the algorithm, and output the running time to the terminal window. You then double the size of this number and repeat this process. After five such increases, there is a set of results from which you can generalize. (Lambert, 2019).

```
"""
File: timing1.py
Prints the running times for problem sizes that double,
using a single loop.
"""
import time
problemSize = 10000000
print("%12s%16s" % ("Problem Size", "Seconds"))
for count in range(5):
    start = time.time()
# The start of the algorithm
    work = 1
    for x in range(problemSize):
        work += 1
        work -= 1
# The end of the algorithm
    elapsed = time.time() - start
    print("%12d%16.3f" % (problemSize, elapsed))
    problemSize *= 2
```

Problem Size	Seconds
10000000	3.8
20000000	7.591
40000000	15.352
80000000	30.697
160000000	61.631

1.2 Runtime

- After running the code, the output window is displayed next to the code. Should you choose to run the code yourself you might get a small variation but the trend is the same. For example in my compiler I had a runtime of 3.305, 4.663, 14.932, and 17.901 seconds respectively. This shows that as the size of the problem doubles so does the execution time; we may safely infer then that a problem of size 320000000 would take 124 seconds.
- Supposing a nested loop is introduced into our program as follows:
- for j in range(problemSize):

  for k in range(problemSize):

  work += 1

  work -= 1
- This loop iterates through the size of the problem within another loop that also iterates through the size of the problem. This program was left running overnight. By morning it had processed only the first data set, 10,000,000. The program was then terminated and run again with a smaller problem size of 1000. that when the problem size doubles, the number of seconds of running time more or less quadruples. At this rate, it would take 175 days to process the largest number in the previous data set! (Lambert, 2019)

1.2 Runtime

- As can be seen from this example there are some drawbacks to using this method; first and foremost different hardware platforms will produce different running times due to processor speed, operating systems and so on. You can already see the difference between my platform and Lambert's!
- For larger datasets it's not practical to use running time as a measure.
- What other options are there to use?
- How about memory, or counting instructions?

1.3 Other measures…

- "Another technique used to estimate the efficiency of an algorithm is to count the instructions executed with different problem sizes. These counts provide a good predictor of the amount of abstract work an algorithm performs, no matter what platform the algorithm runs on." (Lambert, 2019). Unfortunately, after a series of experiments with different codes and counting instructions Lambert (2019) declared that "The problem with tracking counts in this way is that, with some algorithms, the computer still cannot run fast enough to show the counts for very large problem sizes."
- Memory as a measure also presents other challenges: "A complete analysis of the resources used by an algorithm includes the amount of memory required. Once again, focus on rates of potential growth. Some algorithms require the same amount of memory to solve any problem. Other algorithms require more memory as the problem size gets larger. (Lambert, 2019)

Part 2

Complexity analysis

2.1 Introduction

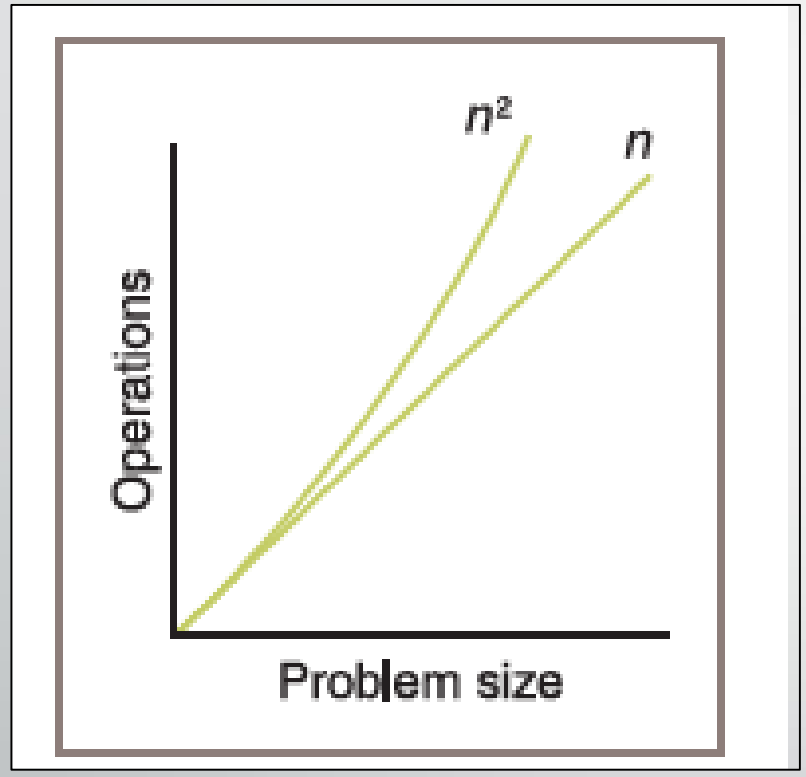
- Definition: "Algorithm complexity is a measure which evaluates the order of the count of operations, performed by a given order algorithm as a function of the size of the input data.
- To put this simpler, complexity is a rough approximation of the number of steps necessary to execute an algorithm.
- When we evaluate complexity we speak of order of operation count, not of their exact count. For example if we have an order of N^2 operations to process N elements, then $N^2/2$ and $3*N^2$ are of one and the same quadratic order." (Svetlin Nakov, 2019)
- Let us demonstrate this using some simple code, algebra, and logic.

2.2 Order of complexity

- We have seen from the example code discussed so far in this lesson that for a problem of size n the first program iterated n times; however, introducing the nested loop increased this to n^2 times. For small values of n the difference is small; but for larger values of n the amount of work done is clearly very different. Figure 1 shows us in graphical form just how divergent the difference is.
- Lambert (2019) observes that "The performances of these algorithms differ by an **order of complexity**. The performance of the first algorithm is **linear** in that its work grows in direct proportion to the size of the problem... The behavior of the second algorithm is **quadratic** in that its work grows as a function of the square of the problem size (problem size of 10, work of 100). As you can see from the graph and the table, algorithms with linear behavior do less work than algorithms with quadratic behavior for most problem sizes *n*. In fact, as the problem size gets larger, the performance of an algorithm with the higher order of complexity becomes worse more quickly."
- Lambert (2019) adds: "An algorithm has **constant** performance if it requires the same number of operations for any problem size. List indexing is a good example of a constant-time algorithm. This is clearly the best kind of performance to have."

Figure 1

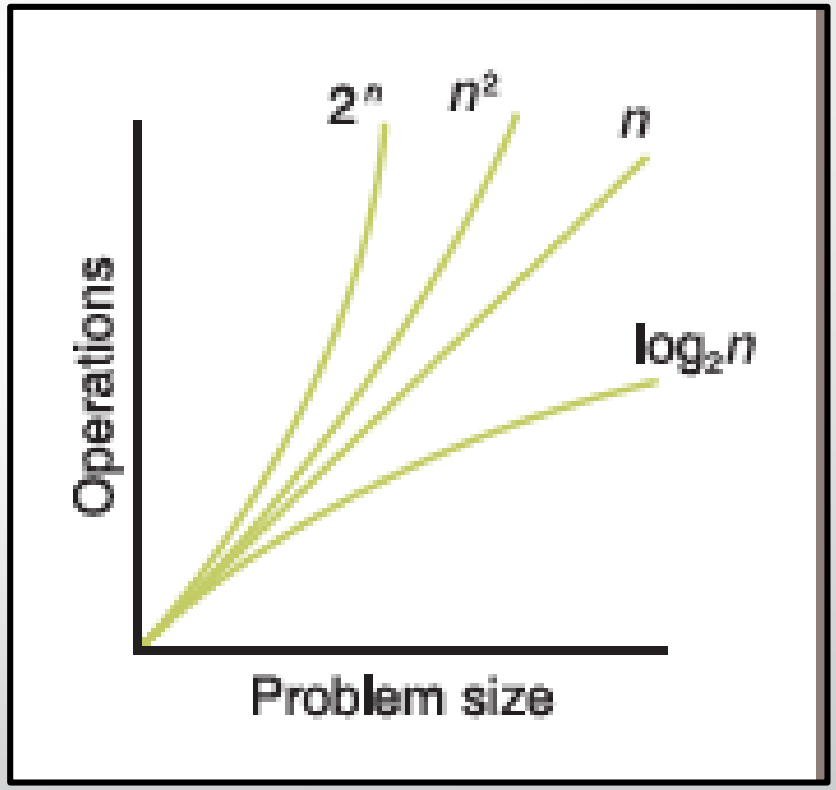
Work done by n and n^2



(From Lambert, 2019)

Figure 2

Sample orders of complexity



(From Lambert, 2019)

2.2 Order of complexity

- Other orders of complexity include:
- Logarithmic: this is better than linear but worse than constant. The amount of work done by this algorithm is proportional to $\log_2$ of the problem size; therefore, when the problem size doubles the amount of work only increases by 1 ($\log_2 2 = 1$).
- Polynomial: these grow at a rate of n^k, where k=1, 2, 3,…. For example n^2, n^3, and so on.
- Exponential: this is worse than polynomial; they are impractical to run with problems of large sizes. An example is $2^{n.}$
- Figure 2 shows the different orders of complexity and their performance as problem size varies. Table 1 gives examples of the orders of complexity.

Table 1

Orders of complexity

Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer' - Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

(From Karumanchi, 2017)

2.3 Asymptotic complexity

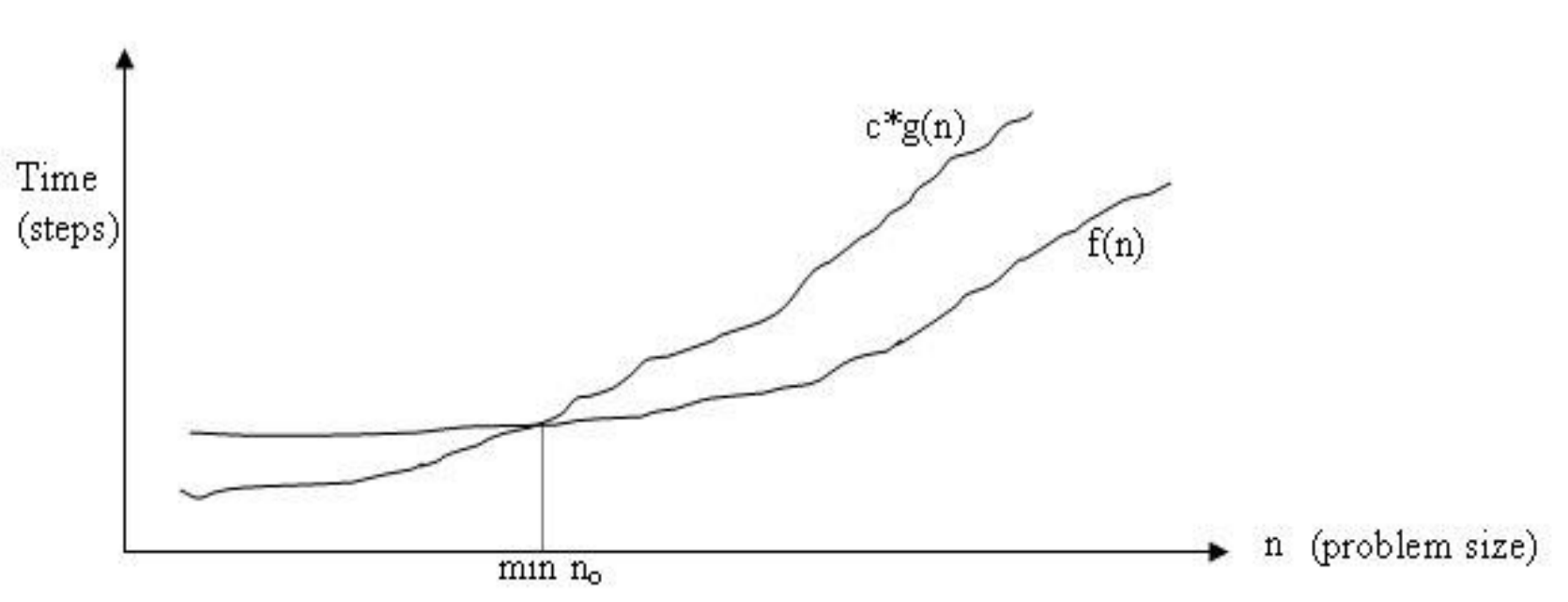
- Let us assume that f(n) is a function that represents a certain algorithm.
- Finding the exact complexity, f(n) = number of basic operations, of an algorithm is a difficult task.
- We approximate f(n) by a function g(n) in a way that does not substantially change the magnitude of f(n). --the function g(n) is sufficiently close to f(n) for large values of the input size n. Recall that our measurement of the efficiency of f(n) is for large n; since we have already seen that for small n the performance differences are small.
- This "approximate" measure of efficiency is called asymptotic complexity.
- Thus the asymptotic complexity measure does not give the exact number of operations of an algorithm, but it shows how that number grows with the size of the input.
- This gives us a measure that will work for different operating systems, compilers and CPUs.

2.4 Big O

- Recall in lesson 1 we introduced best case, average case, and worst case of an algorithm? Summarily the best case defines the input for which the algorithm takes the least time (fastest time to complete); worst case defines the input for which the algorithm takes a long time (slowest time to complete); average case provides a prediction about the running time of the algorithm.
- The Big-O notation, **O(g(n)),** is used to give an upper bound (worst-case) on a positive runtime function f(n) where n is the input size.
- Consider a function **f(n)** that is non-negative $\forall\ n \geq o$. We say that "f(n) is Big-O of g(n)" i.e., f(n) = O(g(n)), if $\exists\ n_o \geq o$ and a constant $c > o$ such that $\mathbf{f(n) \leq cg(n),\ \forall\ n \geq n_o}$
- This is illustrated in figure 3.

Figure 3

Big –O asymptotic function



(Adapted from Black, 2019)

2.4 Big O

- For example, if $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n^4 *is* $g(n)$. That means $g(n)$ gives the maximum rate of growth for $f(n)$ at larger values of n. (Karumanchi, 2017).
- Also $n^2 + 3n + 4$ is $O(n^2)$, since $n^2 + 3n + 4 < 2n^2$ for all $n > 10$ (and many smaller values of n). Strictly speaking, $3n + 4$ is $O(n^2)$, too, but big-O notation is often misused to mean "equal to" rather than "less than" (Black, 2019)
- $g(n)$ is an asymptotic tight upper bound for $f(n)$. Our objective is to give the smallest rate of growth $g(n)$ which is greater than or equal to the given algorithms' rate of growth $/(n)$. Generally we discard lower values of n. That means the rate of growth at lower values of n is not important. In Figure 3, n_0 is the point from which we need to consider the rate of growth for a given algorithm. Below n_0, the rate of growth could be different. n_0 is called threshold for the given function. (Karumanchi, 2017).

2.4 Big O

- Table 2 shows Big-O complexity classes in order of magnitude from smallest to highest, together with some example applications.
- Big-O notation cannot compare algorithms in the same complexity class.
- Big-O notation only gives sensible comparisons of algorithms in different complexity classes when **n** is large . (Necaise, 2011).
- Let us demonstrate this using a simple example that compares a linear and a quadratic function:
- Linear: $f(n) = 1000\,n$; Quadratic: $f'(n) = n^2/1000$
  The quadratic one is faster for $n < 1000000$.
- This means that $f(n) >= f'(n)$ for $n <= 10^6$, and $f'(n) >= f(n)$ otherwise.

Table 2

Big O growth rates

Notation	Type	Examples	Description
O(1)	Constant	Hash table access	Remains constant regardless of the size of the data set
O(log n)	Logarithmic	Binary search of a sorted table	Increases by a constant. If n doubles, the time to perform increases by a constant, smaller than n amount
O(< n)	Sublinear	Search using parallel processing	Performs at less than linear and more than logarithmic levels
O(n)	Linear	Finding an item in an unsorted list	Increases in proportion to n. If n doubles, the time to perform doubles
O(n log(n))	n log(n)	Quicksort, Merge Sort	Increases at a multiple of a constant
$O(n^2)$	Quadratic	Bubble sort	Increases in proportion to the product of n*n
$O(c^n)$	Exponential	Travelling salesman problem solved using dynamic programming	Increases based on the exponent n of a constant c
O(n!)	Factorial	Travelling salesman problem solved using brute force	Increases in proportion to the product of all numbers included (e.g., 1*2*3*4...)

(From Cowan, 2020)

2.4 Big O

- Rules of Big-O (Big O Notation in Data Structure: An Introduction | Simplilearn, 2022):
- Multiplicative Constants Rule: Ignoring constant factors.
- O(c f(n)) = O(f(n)), where c is a constant;
- Example: $O(40\ n^3) = O(n^3)$
- Addition Rule: Ignoring smaller terms
- If O(f(n)) < O(h(n)) then O(f(n) + h(n)) = O(h(n)).
- Example: $O(n^2 \log n + n^3) = O(n^3)$; $O(2000\ n^3 + 2n\ ! + n^{800} + 10n + 27n \log n + 5) = O(n\ !)$
- Multiplication Rule: O(f(n) * h(n)) = O(f(n)) * O(h(n))
- Example: $O((n^3 + 2n^2 + 3n \log n + 7)(8n^2 + 5n + 2)) = O(n^5)$

2.4 Big O

Proving Big O complexity: Going by the definition of Big O, to prove that **f(n)** is **O(g(n))** we find any pair of values $\mathbf{n_0}$ and **c** that satisfy:

$$\mathbf{f(n) \leq c * g(n)} \text{ for } \forall\ \mathbf{n} \geq \mathbf{n_0}$$

Note: The pair $\mathbf{(n_0, c)}$ is not unique. If such a pair exists then there is an **infinite** number of such pairs. (Karumanchi, 2017)

Example: Prove that $\mathbf{f(n) = n^2 + 5}$ is $\mathbf{O(n^2)}$

We try to find some values of n and c by solving the following inequality:

$$\mathbf{n^2 + 5 \leq cn^2} \quad \text{OR} \quad \mathbf{1 + 5/n^2 \leq c}$$

(By putting different values for n, we get corresponding values for c). The output is shown below:

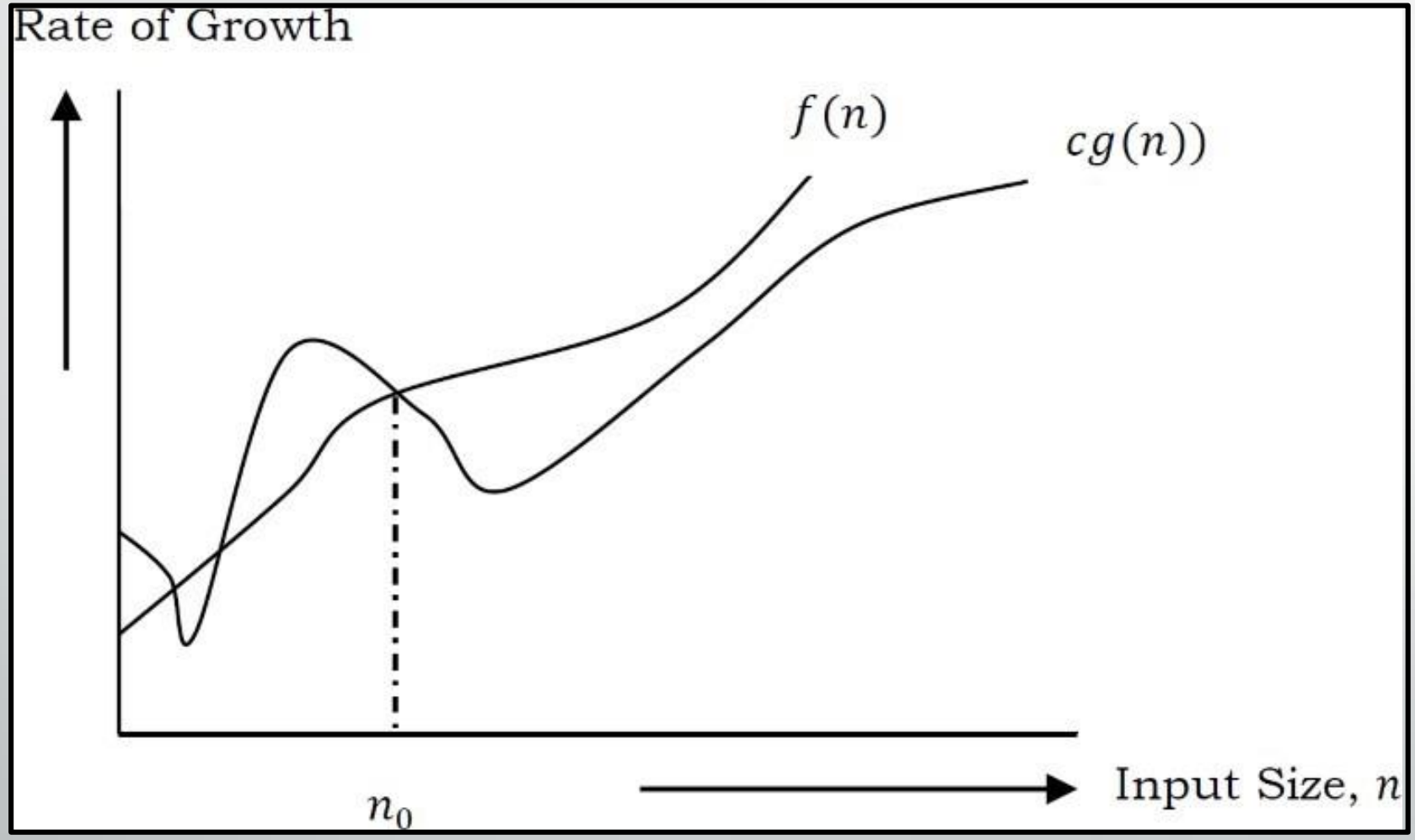
$\mathbf{n_0}$	1	2	3	4	∞
c	6	2.25	1.55	1.3	1

2.5 Big Omega

- Similar to the O discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as $f(n) = \Omega(g(n))$. That means, at larger values of n, the tighter lower bound of $f(n)$ is $g(n)$. For example, if $f(n) = 100n^2 + 10n + 50$, $g(n)$ is $\Omega(n^2)$.
- The function $g(n)$ is $\Omega(f(n))$ iff there exist a real positive constant $c > 0$ and a positive integer n_0 such that $g(n) \geq cf(n)$ for all $n \geq n_0$
- This means that big omega defines the lower bound while big o defines the upper bound. Consequently, If $f(n)$ is $O(g(n))$ then $g(n)$ is $\Omega(f(n))$
- Figure 4 shows the big omega function.
- Let us discuss an example problem: Find lower bound for $f(n) = 5n^2$.
- Solution: $\exists\ c, n_0$ Such that: $0 \leq cn^2 \leq 5n^2 \Rightarrow cn^2 \leq 5n^2 \Rightarrow c = 5$ and $n_0 = 1$
- $\therefore 5n^2 = \Omega(n^2)$ with $c = 5$ and $n_0 = 1$

Figure 4

Big Omega



(From Karumanchi, 2017)

2.6 Theta

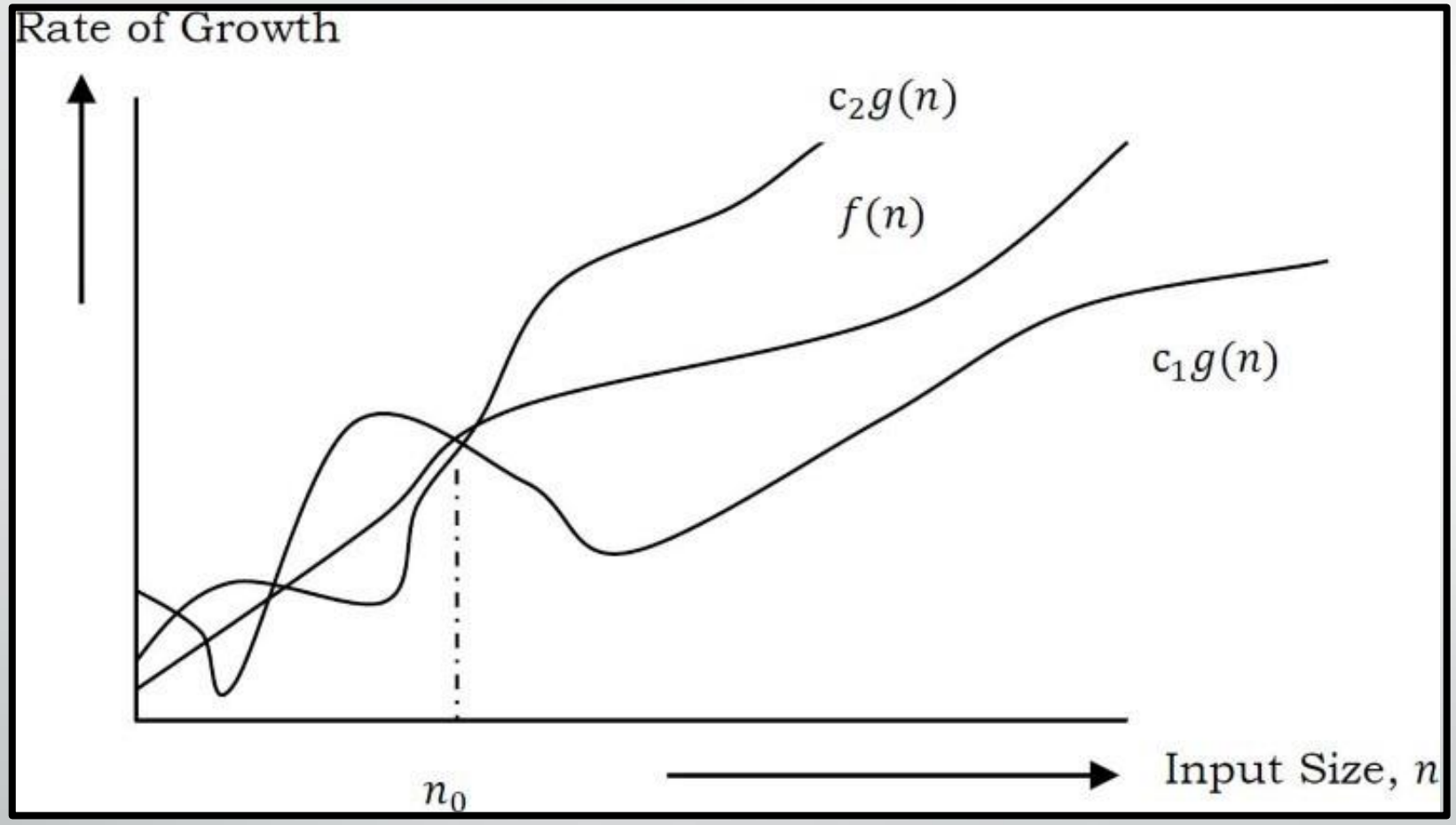
- The function $g(n)$ is $\Theta(f(n))$ iff there exist two real positive constants $c_1 > 0$ and $c_2 > 0$ and a positive integer n_0 such that:

  $$c_1 f(n) \geq g(n) \geq c_2 f(n) \text{ for all } n \geq n_0$$

  - Whenever two functions, f and g, are of the same order, $g(n)$ is $\Theta(f(n))$, they are each Big-Oh of the other: $g(n)$ is $O(f(n))$ AND $f(n)$ is $O(g(n))$
- Figure 5 presents the two functions as they relate to the constants.
- Let us consider an example: Prove $n \neq \Theta(n^2)$
- **Solution:** $c_1 n^2 \leq n \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq 1/c_1$
- $\therefore n \neq \Theta(n^2)$
- Figure 6 captures all three bounds showing how they are related.

Figure 5

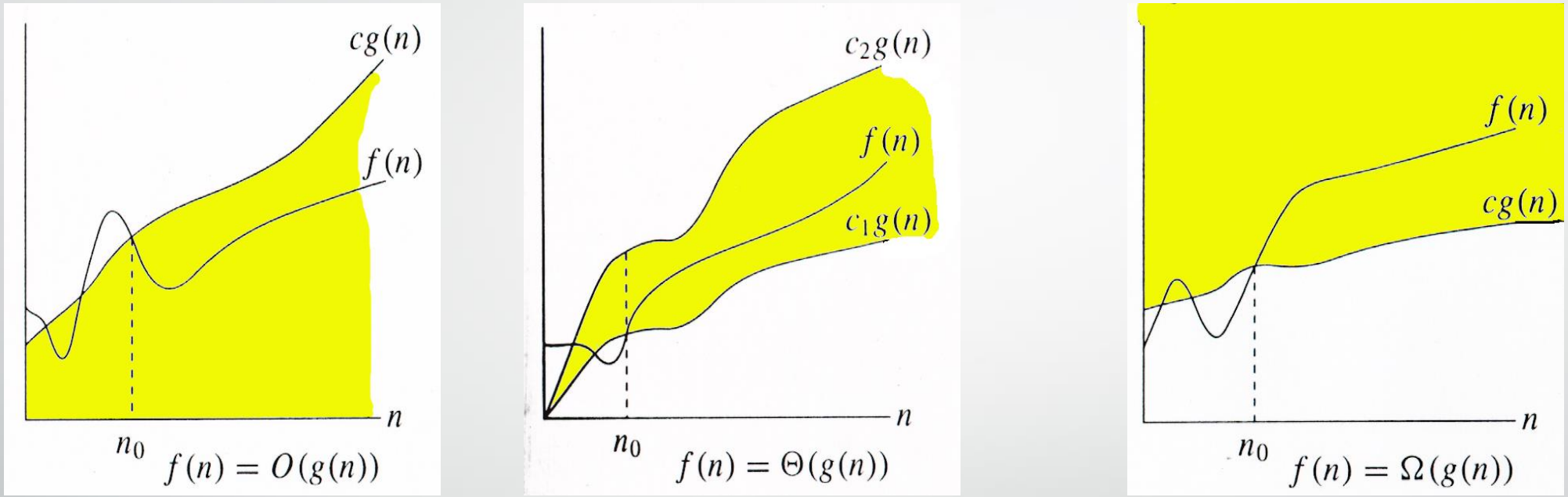
Theta function



(From Karumanchi, 2017)

Figure 6

Relation between Big O, Theta, and Omega



(From Karumanchi, 2017)

2.7 Complexity in code

- Most of the time you would also like to determine the complexity of code that you have written from the algorithm that solves the problem.
- Program codes also have their big O computed just like has been done in this lesson. The idea is to determine the manner in which the program executes; whether it is linear, or a loop, iteration, a nested loop, and so on.
- There are some rules that are followed when computing the complexity of code. We examine some program codes and determine their complexity in the next few slides. These guidelines and examples are all from Karumanchi (2017).

2.7 How to determine complexity of code structures

Loops: for, and while:

Complexity is determined by the number of iterations in the loop times the complexity of the body of the loop.

Examples:

```
for (i = 0; i < n; i++)
      sum = sum - i;
```

$O(n)$

```
for (i = 0; i < n * n; i++)
      sum = sum + i;
```

$O(n^2)$

```
i=1;
while (i < n):
      sum = sum + i;
      i = i*2
```

$O(\log n)$

2.7 How to determine complexity of code structures

Nested Loops: Complexity of inner loop * complexity of outer loop.

Examples:

```
sum = 0
For(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
        sum += i * j ;
```

$O(n^2)$

```
i = 1;
while(i <= n):
   j = 1;
   while(j <= n):
      #statements of constant complexity
      j = j*2
      i = i+1
```

$O(n \log n)$

2.7 How to determine complexity of code structures

Sequence of statements: Use Addition rule

$O(s1; s2; s3; \ldots sk) = O(s1) + O(s2) + O(s3) + \ldots + O(sk)$

$= O(\max(s1, s2, s3, \ldots, sk))$

Example:

```
for (j = 0; j < n * n; j++)
    sum = sum + j
for (k = 0; k < n; k++)
    sum = sum - 1
print("sum is now " + sum)
```

Complexity is $O(n^2) + O(n) + O(1) = O(n^2)$

2.7 How to determine complexity of code structures

If Statement: Take the complexity of the most expensive case **:**

If (……some code) **$O(n^2)$**

Elif (……some code) **O(n3)**

(……some code) **O(1)**

Overall complexity **O(n3)**

2.7 How to determine complexity of code structures

- Sometimes if-elif statements must carefully be checked:

O(if-elif) = O(Condition)+ Max[O(if), O(elif)]

```
int[] integers = new int[10];
.........
if(hasPrimes(integers) == true)
     integers[0] = 20;
else
     integers[0] = -20;


public boolean hasPrimes(int[] arr) {
      for(int i = 0; i < arr.length; i++)
                  ..........
                  ..........
} // End of hasPrimes()
```

O(1)

O(1)

O(n)

$$O(\text{if-elif}) = O(\text{Condition}) = \boldsymbol{O(n)}$$

2.7 How to determine complexity of code structures

- **Note:** Sometimes a loop may cause the if-else rule not to be applicable. Consider the following loop:

```
while (n > 0):
  if (n % 2 = = 0):
    print(n)
    n = n / 2
  else:
    print(n)
          //print (n+2)
    n = n – 1
```

The else-branch has more basic operations; therefore one may conclude that the loop is O(n). However the if-branch dominates. For example if n is 60, then the sequence of n is: 60, 30, 15, 14, 7, 6, 3, 2, 1, and 0. Hence the loop is logarithmic and its complexity is O(log n)

2.7 How to determine complexity

- What is the complexity of the program given below (Karumanchi, 2017):

```
def Function(n):
    count = 0
    for i in range(n/2, n):
        j = l
        while j + n/2 <= n:
            k = l
            while k <= n:
                count= count+ 1
                k = k * 2
            j = j + 1

    print (count)
Function(20)
```

Solution:

```
def Function(n):
    count = 0
    for i in range(n/2, n):                    #Outer loop execute n/2 times
        j = 1
        while j + n/2 <= n:                    #Middle loop executes n/2 times
            k = 1
            while k <= n:                      #Inner loop execute logn times
                count = count + 1
                k = k * 2
            j = j + 1

    print (count)

Function(20)
```

The complexity of the above function is $O(n^2 logn)$.

NB// O(n/2) can be reduced to O(n). In other words , the complexity of looking at half the elements is no worse than looking at all the elements

What is the complexity of the code below (Karumanchi, 2017):

```
def Function(n):
        count = 0
        for i in range(n/2, n):
                j = 1
                while j + n/2 <= n:
                        k = 1
                        while k <= n:
                                count = count + 1
                                k = k * 2
                        j = j * 2

        print (count)

Function(20)
```

Solution:

```
def Function(n):
    count = 0
    for i in range(n/2, n):                    #Outer loop execute n/2 times
        j = 1
        while j + n/2 <= n:                    #Middle loop executes logn times
            k = 1
            while k <= n:                      #Inner loop execute logn times
                count = count + 1
                k = k * 2
            j = j * 2
    print (count)

Function(20)
```

The complexity of the above function is $O(nlog^2n)$.

Part 3

Search algorithms

3.1 Sequential Search

- The sequential search looks at each item at a time. If it is present its position is returned.
- If it is missing, something, e.g. -1, is returned to signal that the item was not found.
- Sequential search is also known as linear search.
- It is easy to code but not very efficient.

```
def search(x, nums):
        for i in range(len(nums)):
                if nums[i] == x:
        #found
                        return i
        return -1 #loop finished, not
found
```

3.1.1 Analysis of Sequential Search

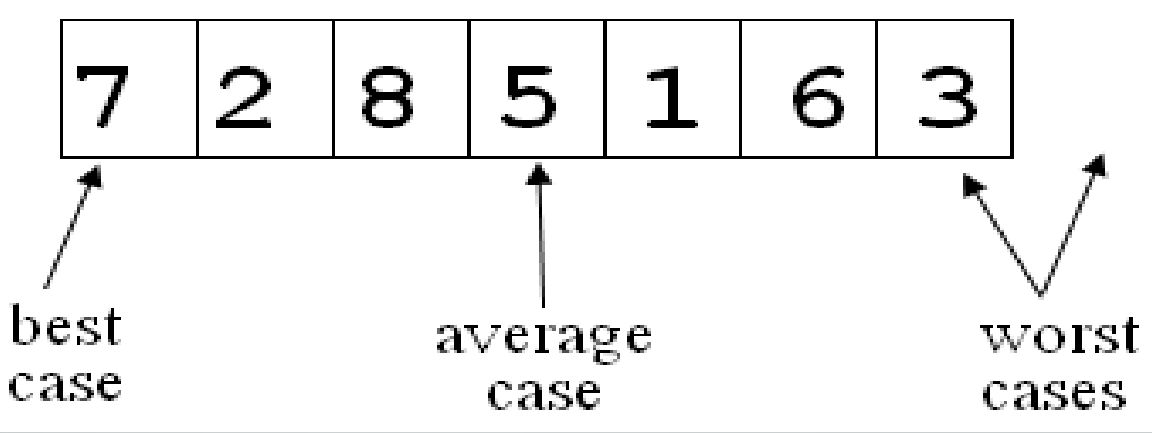
- Sequential search will work very nicely for modest-sized lists.
- But it will prove to very inefficient for large lists.
- However, it can't be helped if the list is unsorted.
- Imagine if your textbook did not have a table of contents or chapter headings. Then when you need to look up a topic, you start from page one and flip through until you find it.
- If that topic is not covered, you have to look all the way to the last page.

3.1.1 Analysis of Sequential Search

- For unordered lists, one has no choice but to use sequential search
- We call this an **order n** search, meaning that in the worst case, if the list has n items, the loop runs n times.
- We write **O(n)** to indicate order n.
- This is called the **Big-O** notation and the discipline of analyzing algorithm efficiency is known as **computational complexity**.
- In complexity analysis, we want to express the time an algorithm takes to solve a problem as a function of the problem size.

3.1.2 Worst Case Complexity

- In complexity analysis, we are mostly interested in the worst case performance of the algorithm.
- The worst case in sequential search is when the item is either at the end or not in the list.
- In this case the loop runs to the end, taking the maximum number of steps, **n,** as seen in the example array below.



(From Lambert, 2014)

3.1.3 Complexity Analysis

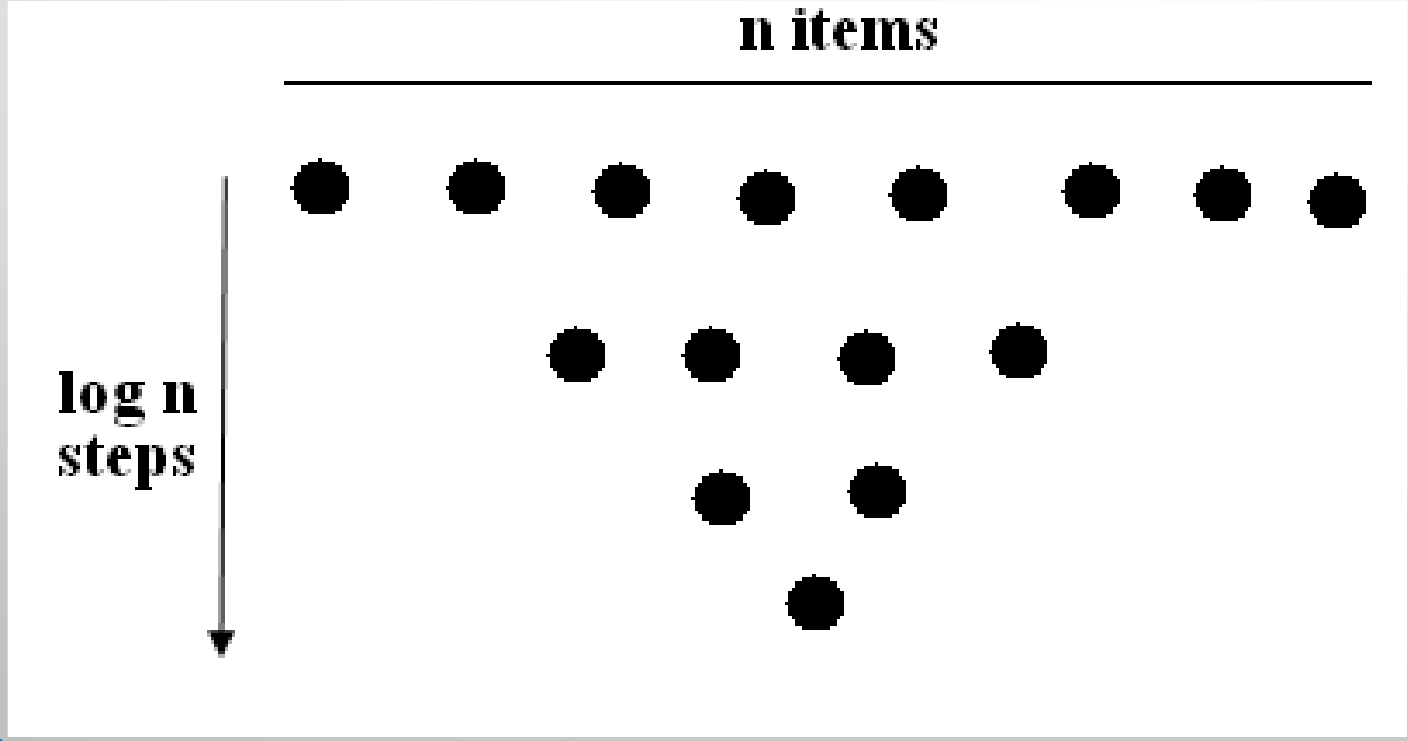
- This example emphasizes the importance of complexity analysis, as it allows us to compare algorithms that perform the same task, e.g. two different search algorithms.
- Why not just implement them and run them on a computer and see which one is faster? There are a few problems with this approach.
  - Different hardware gives different results
  - Different operating systems give different results
  - Different programming languages give different results (e.g. C is much faster than Python)
  - If a program takes year on a given large problem, we would have to wait a year to decide that the algorithm is no good.

3.2 Binary Search

- If we have a very large collection of data, we might want to organize it in some way so that we don't have to look at every single item.
- If the list is sorted, e.g., the phone book, we can use binary search.
- Binary search is said to be **O(log n)**, which is **much faster** than **O(n)**.
- Binary search uses **repeated halving** to achieve this efficiency.

3.2.1 Repeated Halving

- Starting from some integer **n**, if **n** is repeatedly halved, how many iterations must be applied to reduce **n** to 1?
- The answer is $\log_2 n$ iterations.

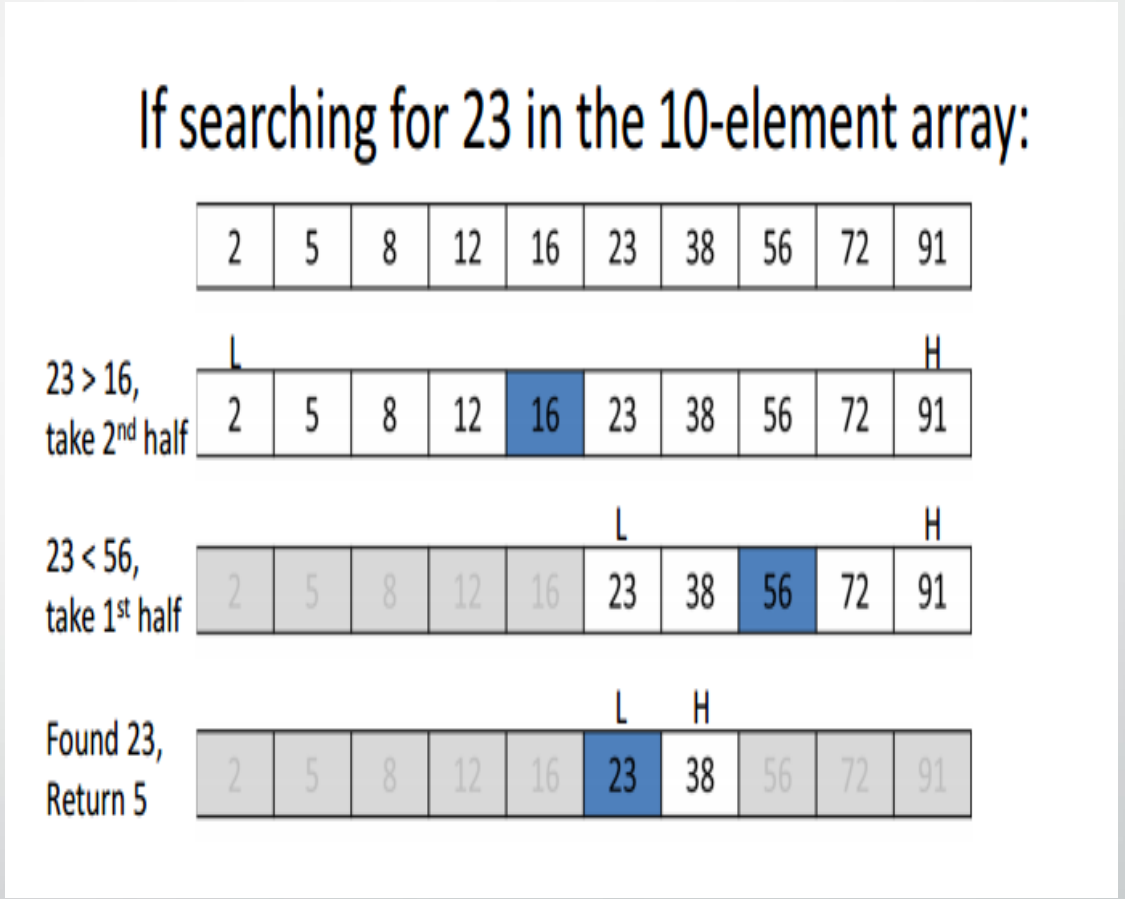


3.2.1 Repeated Halving in Binary Search

- The binary search algorithm uses the repeated halving technique to search the list.
- First, the search item is compared with the middle element.
- If the search item is found, the search terminates.
- If the search item is less than the middle element of the list, we restrict the search to the first half; otherwise, we search the second half.
- In each iteration the candidate elements are half the previous number.
- The worst case for the binary search algorithm is $O(\log_2 n)$.
- Table 3 demonstrates the efficiency of this method with different list sizes. (Lambert, 2014)

Divide and Conquer Algorithm for Binary Search

1. Create two variables called **low** & **high** initialized to 0 & array length minus 1.
2. Repeat while (low <= high)
   a) Calculate **mid**, a point between them (rounding down to the nearest unit), and check the value stored at that location.
   b) If it is a match, exit and return **mid**.
   c) If the target value is smaller than **A[mid]** set **high** to **mid-1**.
   d) If the target value is bigger than **A[mid]** set **low** to **mid+1**.
3. The item is not in the array.



(From gracekstateengg, 2017)

Binary Search Function

```
def search(x, nums):
        low = 0
        high = len(nums) - 1
        while low <= high:
                mid = (low + high)//2
                item = nums[mid]
                if x == item:
                        return mid
                elif x < item:
                        high = mid - 1
                else:
                        low = mid + 1
        return -1
```

Repeated Halving Efficiency

Table 3

Binary search efficiency

List Size	Halvings
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
1,048,576	20

(From Lambert, 2014)

Part 4

Sort algorithms

4.1 Introduction

- Sorting takes a list and rearranges the values so that they are in increasing (or decreasing) order.
- Sorting makes searching easier:
- Names in the phone book are sorted; otherwise finding a name would be very difficult.
- You may need to delete all mail from a particular spammer. Sorting makes this task easy.
- You may need to see who the top scorers in a game are: sort in decreasing order.

4.1 Efficiency in Sorting

- As we did with searching, we will look at two categories of sorting:
  1. Relatively easy to code but not very efficient. These are called **sequential sorts**. Examples are: selection sort, insertion sort, bubble sort
  2. Rather complex to code but very efficient. These are called **logarithmic sorts**. Examples are merge sort and quick sort.
- Efficiency only matters when the amount of data to be processed is large.
- For small lists, just about any algorithm will do.

4.2 Selection Sort

- The steps to be followed are:
- Scan the entire list to find the smallest value, $\mathbf{s_0}$
- Exchange that value with the value in position **0** of the list.
- At this point $\mathbf{s_0}$ is sorted.
- But the remainder of the list is not. Find the smallest value $\mathbf{s_1}$ and exchange it with whatever is in position **1**.
- Continue this way until the entire list is sorted. This is demonstrated in Table 4.; the starred items have just been swapped, and the highlighted portion is sorted.

Table 4

Selection sort

Unsorted	After 1st Pass	After 2nd Pass	After 3rd Pass	After 4th Pass
5	1*	1	1	1
3	3	2*	2	2
1	5*	5	3*	3
2	2	3*	5*	4*
4	4	4	4	5*

(From Lambart, 2014)

4.2.1 Swapping

- Since sorting will likely involve values swapping their positions, we define a swap function as follows:

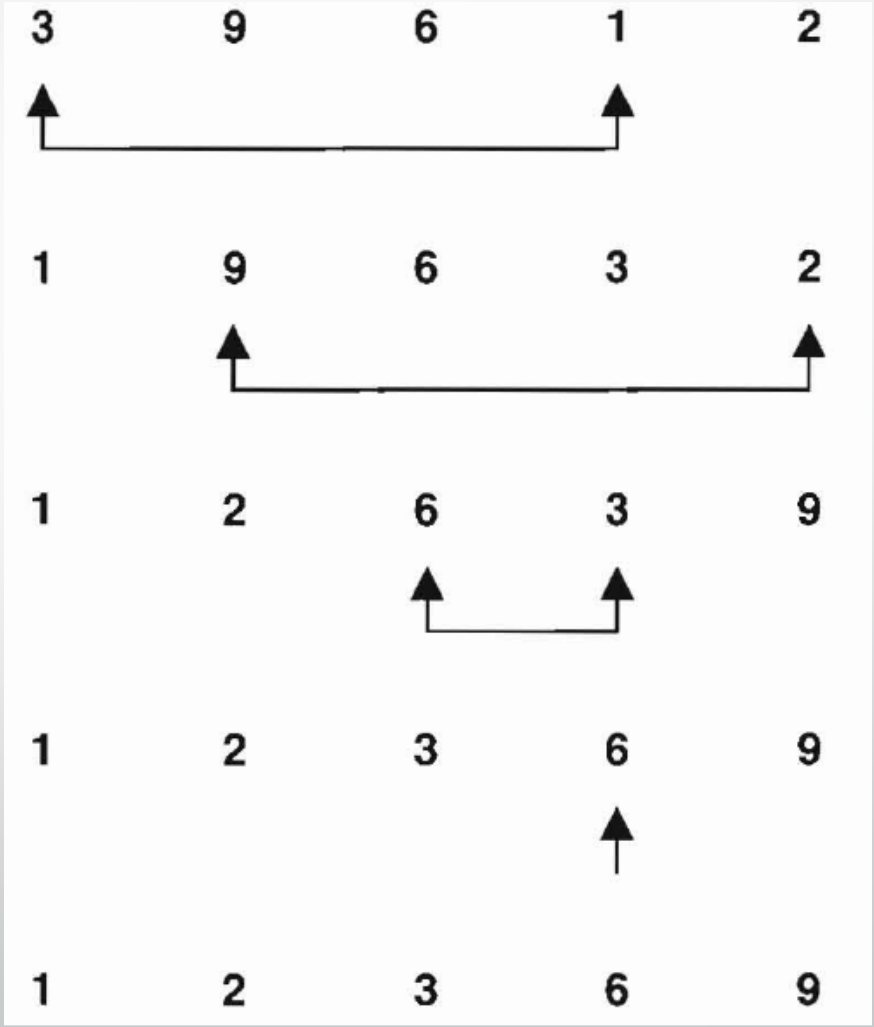
```
def swap(lst, i, j):
# You could write lst[i], lst[j] = lst[j], lst[i]
# But the following code shows what is really going on
      temp = lst[i]
      lst[i] = lst[j]
      lst[j] = temp
```

Selection Sort Function

```
def selectionSort(lst):
  i = 0
  while i < len(lst) - 1:
              minIndex = i
              j = i + 1
  while j < len(lst):
  if lst[j] < lst[minIndex]:

       minIndex = j
              j = j + 1
  if minIndex != i:

       swap(lst, minIndex, i)
              i = i + 1
```



(From Mbogho, 2018)

4.2 Analysis of Sequential Sorts

- Selection sort is relatively simple but inefficient.
- It uses a pair of nested loop, which implies roughly $\mathbf{n^2}$ steps in the worst case.
- We call it an **order n^2,** that is $O(n^2)$ algorithm.
- This is fine for small lists, but sorting 10,000 items would (in the worst case) require 100,000,000 steps.

4.3 Merge Sort

- Merge sort belongs to a class of algorithms known as divide and conquer (like binary search)
- That is, you reduce the problem in some way and solve the smaller, easier problem.
- You eliminate the unnecessary work that less efficient algorithms do.
- This makes it a more efficient algorithm.

4.3.1 Sorting a Deck of Cards

- Suppose you and a friend want to sort a deck of cards.
- Split the deck in half so that each person sorts his/her half.
- Combine the two sorted halves (somehow)
- A list of numbers can be sorted the same way.
- This is achieved by splitting the list in a systematic way.

4.3.1 Splitting

- Splitting can be done using Python's slicing mechanism.

  nums = [5, 3, 9, 6, 4, 8]

- Calculate the midpoint

  m = len(nums) // 2

- nums[:m] gives the first half of the list

  [5, 3, 9]

- nums[m:] gives the second half of the list

  [6, 4, 8]

4.3.1 Merging

- Assuming the two halves are sorted, each has its smallest value on top.
- To merge, see which of the two top values is the smaller and make it the first item of the merged list
- Repeat until one of the lists runs out.
- Put all items in the remaining list in the merged list.
- Exercise: merge the two lists below.

5	9	12	20

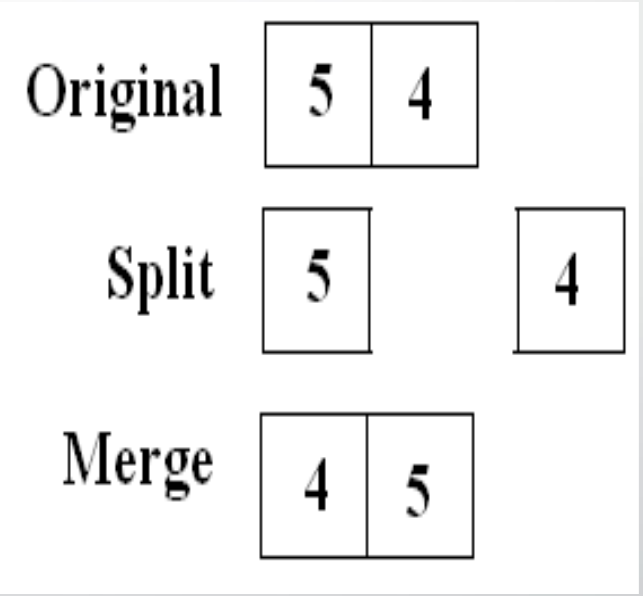
4	16	99	100	560

4.3 Merge Sort

- We know how to split a list in the middle
- We know how to put together two sorted lists into one sorted list
- But how did the halves become sorted
- It turns out, if we keep halving the list repeatedly until we have **lists of length 1**, we never have to actually sort!
  - We can just merge (repeatedly) and we're done.
- The mergesort Python code demonstrates this; it is like splitting 5 and 4, sorting them out and then merging the sorted list.
- We demonstrate this using an example list in Figure 7

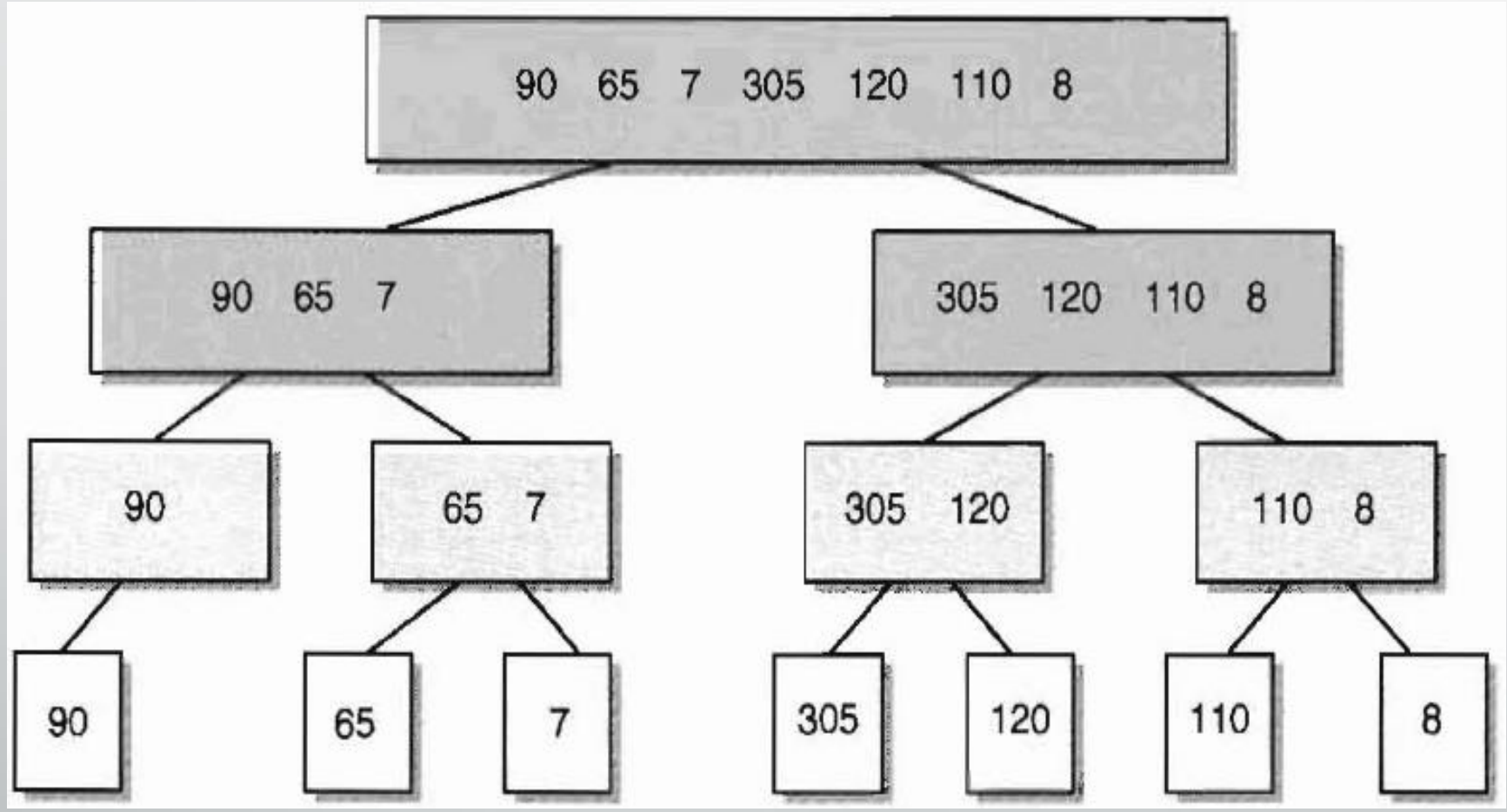
Merge Sort Function

```
def mergeSort(nums):
    n = len(nums)
    # do nothing if nums contains 0 or 1 items
    if n > 1:
        # split into two sublists
        m = n // 2
        nums1, nums2 = nums[:m], nums[m:]
        # recursively sort each piece
        mergeSort(nums1)
        mergeSort(nums2)
        # merge the sorted pieces back into original list
        merge(nums1, nums2, nums)
```



(From Mbogho, 2018)

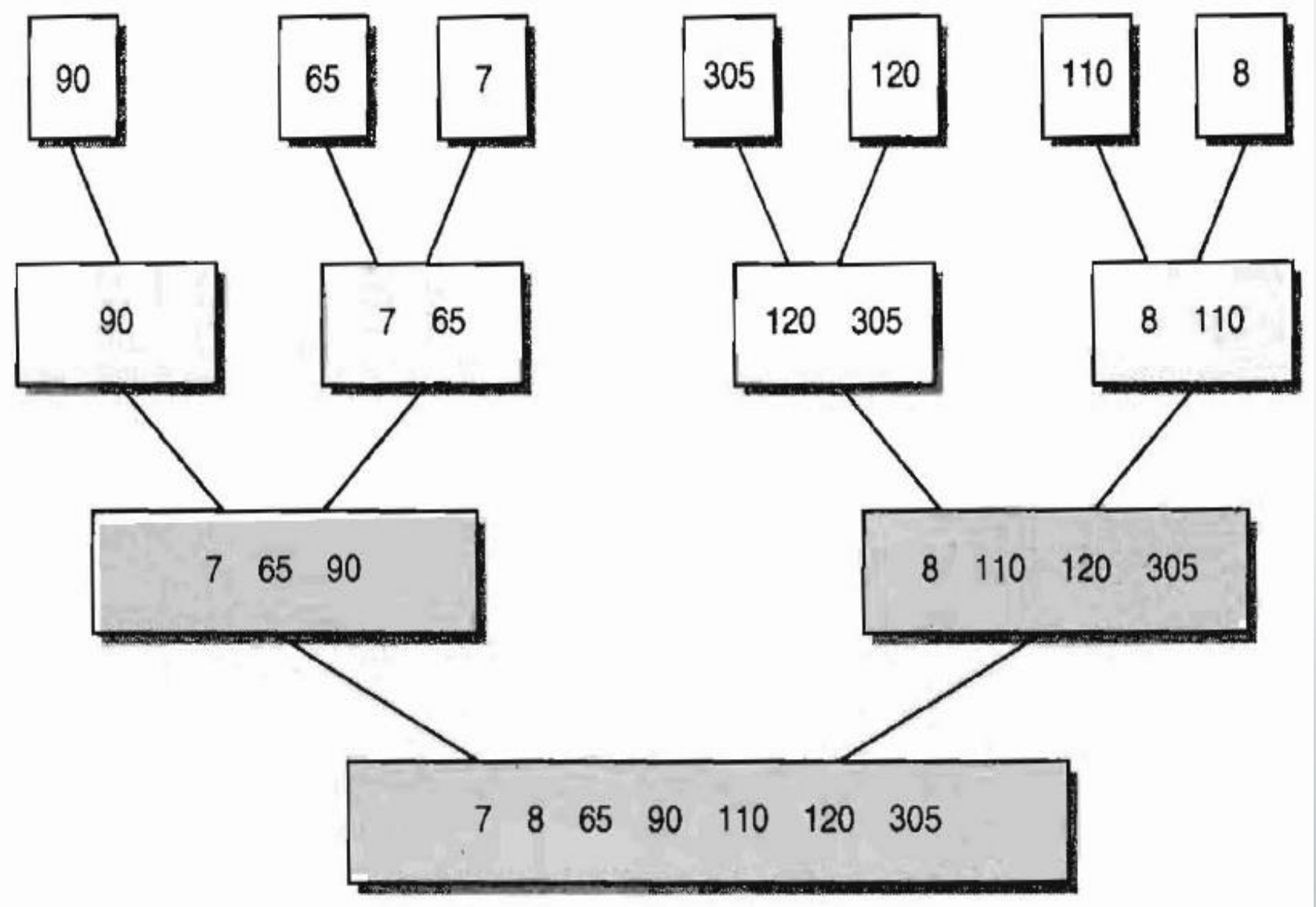
Figure 7 (a)
Splitting and Merging Illustration



(From Mbogho, 2018)

Figure 7 (b)

Splitting and merging illustration



(From Mbogho, 2018)

4.4 Analysis

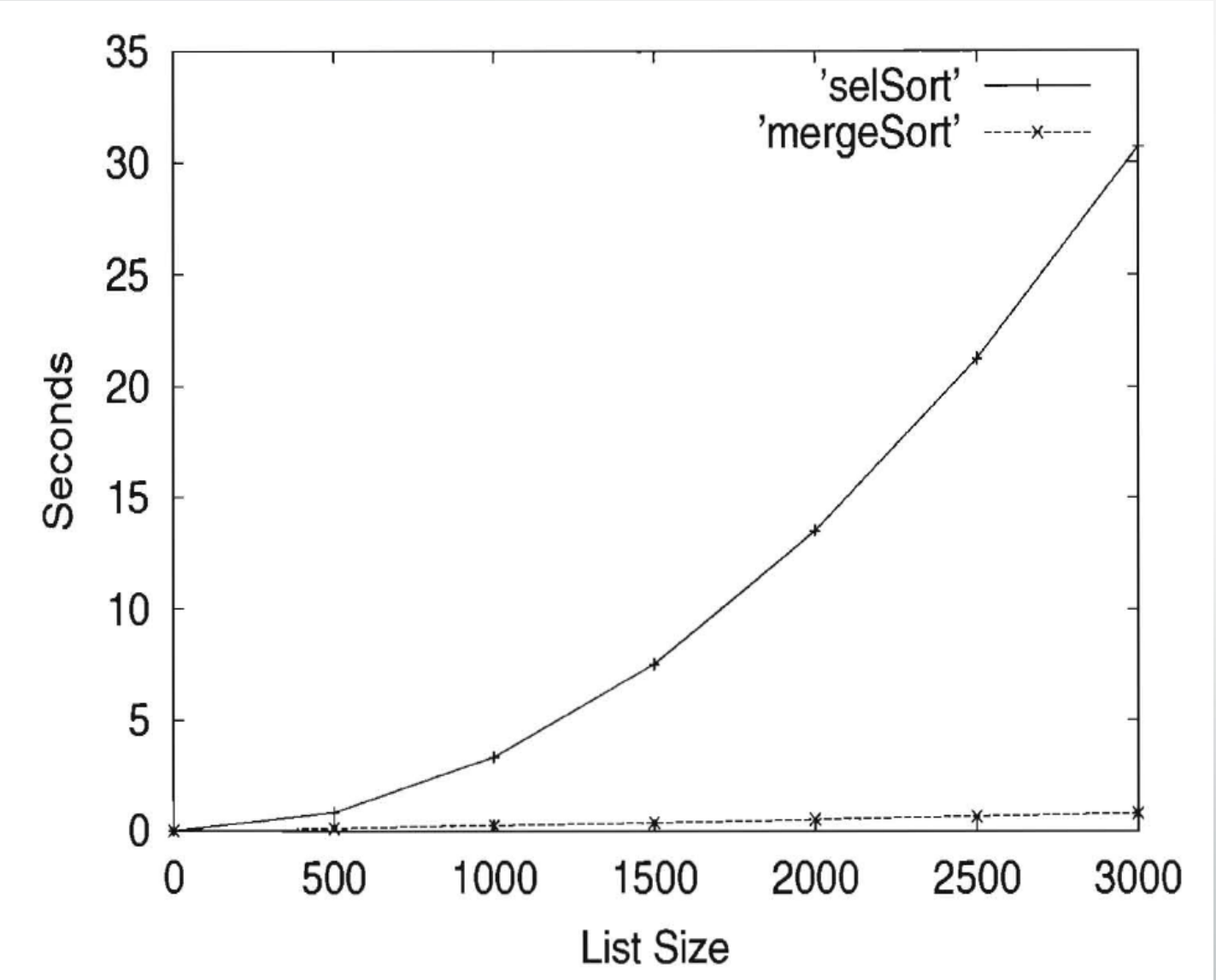
- **Selection sort:**
- Let's take another look at selection sort.
- In order to find the smallest value, the algorithm has to inspect each of the n-1 items.
- The second time around, it has to inspect n-2 items.
- The third time, n-3 items, and so on.
- Hence, the total number of iterations is $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = (\mathbf{n^2}-n)/2$
- This is $\mathbf{O(n^2)}$
- We only need to retain the **dominant** term because for large problems, the other terms become insignificant. (Mbogho, 2018)

4.4 Analysis

- **Merge sort:**
- Merge sort halves repeatedly.
- We know that repeated halving takes log **n** operations.
- But for each halving, the merging copies all **n** items
- Therefore merge sort is **O(n log n)**
- This is far better than **O(n^2)**
- Figure 8 shows the result in graphical form

Figure 8

Selection sort vs merge sort experiment



(From Mbogho, 2018)

Summary

- Algorithm complexity is a measure which evaluates the order of the count of operations, performed by a given order algorithm as a function of the size of the input data.
- The asymptotic complexity measure does not give the exact number of operations of an algorithm, but it shows how that number grows with the size of the input.
- The Big-O notation, **O(g(n)),** is used to give an upper bound (worst-case) on a positive runtime function f(n) where n is the input size.
- The big omega gives the tighter lower bound of the given algorithm and we represent it as $f(n) = \Omega(g(n))$.
- Sequential search will work very nicely for modest-sized lists, but it will prove to very inefficient for large lists. It is O(n) complexity.
- Binary search is said to be **O(log n)**, which is **much faster** than **O(n)**.
- Selection sort is **$O(n^2)$** algorithm
- Merge sort is **O(n log n); t**his is far better than **$O(n^2)$**

References

- *Big O Notation in Data Structure: An Introduction | Simplilearn*. (2022, September 1). Simplilearn.com. https://www.simplilearn.com/big-o-notation-in-data-structure-article#:~:text=What%20are%20the%20rules%20of
- Black, P. E. (2019, September 6). *big-O notation*. Xlinux.nist.gov; dictionary of algorithms and data structures. https://xlinux.nist.gov/dads/HTML/bigOnotation.html
- Cowan, D. (2020). *Big O Notation*. The Science of Machine Learning. https://www.ml-science.com/big-o-notation
- gracekstateengg. (2017, September 12). *Binary Search Algorithm*. Adventures of CompSci; gracekstateengg. https://gracekstateengg.wordpress.com/2017/09/12/binary-search-algorithm/
- Lambert, K. (2014). *Fundamentals of Python*. Cengage Learning Ptr.

References

- Lambert, K. (2019). *Fundamentals of Python: Data Structures*. Cengage Learning.
- Mbogho, A. (2018). *Sorting and Complexity* [Internet to John Mbogholi].
- Sharing of notes for editing for advanced programming
- Narasimha Karumanchi. (2017). *Data structures and algorithms made easy : to all my readers : concepts, problems, interview questions*. Careermonk Publications.
- Necaise, R. D. (2011). *Data structures and algorithms using Python*. John Wiley And Sons.
- Svetlin Nakov. (2019). *Introduction to Programming with C# / Java Books» Chapter 19. Data Structures and Algorithm Complexity*. Introprogramming.info. https://introprogramming.info/english-intro-csharp-book/read-online/chapter-19-data-structures-and-algorithm-complexity/