COMP3506  
Master Notes

SYED MUHAMMAD ZAHIR

# 

# Introduction

The following are a heap (no pun intended) of notes which I have compiled to summarise content learned in COMP3506. This document is free to view, comment, expand upon, modify, download and distribute. Its sole purpose is to make learning Algorithms & Data Structures to be a fun and easy course to grasp. However, I do suggest that in keeping the document in its shape and the notes orderly and clean, that any edits be done so in *Suggesting* mode and that *Comments* on work are preferred over simply writing over it. Those are the only two rules I put forward.

In the process of making this document, I have learnt a lot and want to impart my knowledge which I have gained from a trillion YouTube gurus, the excellent tutors in the course, the lectures and more.

I will have made many mistakes, so please forgive them, and instead comment, edit and modify away for the benefit of this document and others that may wish to use this.

One valuable lesson which I have learnt is that knowledge should be spread at the cost of sharing it to others who may benefit from it. So, please do the same!

# Discussion

| Logs |
| --- |
| **Syed Muhammad 10:42 14/11:**  I have stopped contributing to this document as I was the sole writer and due to lack of time. It is nearing the exam day! To future viewers, go ahead and carry on what I couldn't finish. |

# 1 Algorithm Analysis

| Terms & Definitions | |
| --- | --- |
| *Running time/time complexity* | Running time is how long it takes for an algorithm to execute. In the case of algorithms, we generally talk about running time expressed as a function of input size. |
| *Space complexity* | Space complexity is the amount of memory required in the process of an algorithm’s execution. This is also generally described as a function of the input size. |
| *Pseudo-code* | A language used to detail the structure and logic of certain code without language specificity and strict emphasis of syntax. |
| *Primitive operations* | Operations that are usually described as simple or fundamental in the programming sense, such as variable assignment, conditional checking, equality checking, variable incrementing/ decrementing, basic arithmetic operations, etc. These operations can also be described as the building blocks of lowest level programming languages, i.e., such used in Assembly Language. |
| *Asymptotic analysis* | Asymptotic analysis is the process of defining the mathematical bounding of an algorithm based on its run-time performance. |
| *Big-O notation* | Big-O notation defines the mathematical upper-bound of an algorithm's performance (i.e., either space or time complexity). |
| *Big-Theta notation* | Big-Theta notation defines the mathematical tight-bound of an algorithm's performance (i.e., either space or time complexity). |
| *Big-Omega notation* | Big-Omega notation defines the mathematical lower-bound of an algorithm's performance (i.e., either space or time complexity). |

## 1.1 Pseudo-code

This course introduces a systematic and consistent way to write pseudo-code however, while it's the provided method, it is not enforced as long as the code is readable, understandable and consistent.

The following structure describes the pseudo-code:

| **Method declaration**   | Algorithm *methodName*(args...)Input: **<description of each arg>**Output: **<description of output>** | | --- |   **Variable assignment**   | variableName <- **<value>** | | --- |   **Comparison checking**   | variable1 = variable2variable1 < variable2variable1 > variable2 variable1 <= variable2 variable1 >= variable2 | | --- |   **Mathematical formatting**   |  | | --- |   **Return statement**   | return **<value>** | | --- |   **Control flow**  *Simple boolean check:*   | if **<boolean condition>** then ... | | --- | | *If-then-else:*   | if **<boolean condition**> then ...else ... | | --- |   *Multiple boolean checks:*   | if **<boolean condition>** then ...else if **<boolean condition>** then ......else ... | | --- |   *For-to-do:*   | for **<iterator> to <max>** do  ... | | --- |   *While-do:*   | while **<boolean condition>** do  ... | | --- |   *Repeat-until:*   | repeat  ... until **<boolean condition>** | | --- | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## 1.2 Counting Primitive Operations

Primitive operations really are language-specific (meaning that they depend on what programming language you use) and it comes down to what your code eventually compiles to - which is, *machine code*! However it is quite intuitive to identify the building blocks and perform counting of primitive operations on provided code. Each of the following represents 1 count of primitive operation.

**Variable Assignment**

| abc <- 1 |
| --- |

The machine is allocating memory to store 1 with a pointer to that block of memory stored in abc.

**Comparison Checking**

| abc >= 1 |
| --- |

While comparisons are eventually bitwise and that depends on the size of the input, we will assume that boolean checking is one primitive operation.

**BasicArithmetic Operations**

| abc <- 1 abc + 1 |
| --- |

Basic arithmetic operations like addition, subtraction, multiplication, division, modular operations and the like are all assumed to be a single primitive operation.

**Incrementing/Decrementing**

| abc++ |
| --- |

Incrementing a number is usually dependent on how efficient the compiler is, that is, if it recognises that a machine code exists which increments the value and keeps the same pointer attached to the variable, and one that does not do the following equivalent but slower operation.

| abc <- abc + 1 |
| --- |

Note that there is first on the right hand side an addition operation and then followed by an assignment, which shows that in this case, incrementing/decrementing a number is actually two primitive operations - *not one*! However, with an appropriate assumption, you can get away with either - at least I hope.

**Indexing**

| abc <- [1, 0, 5] abc[0] // accessing/indexing counts as a single primitive operation |
| --- |

Again, we assume that the computer is a magician and knows where everything is and can get a value from a specific pointer address in no less than any one primitive operation.

However, when using control flow operations such as loops, more than one primitive operation is done as shown in the following examples:

**Counting for loop operations**

| for i <- 0 to 5 do  ... |
| --- |

This for-to-do loop shows that there is an initial variable assignment and there are some hidden primitive operations which can only be observed in language-specific cases. The following is the same above for-to-do loop implemented in a Java- or C-like programming language.

| // Java-ish implementation of the above for-to-do loopfor (int i = 0; i < 5; i++) { ... // body} |
| --- |

As you can see the assignment happens at the start of the loop, then before each time the body is executed, a conditional check is done on i. Then after each time the body is executed, i is incremented. Thus excluding the body, there is variable assignment, conditional checking and incrementing of the variable. The assignment occurs once, the conditional check happens before everytime the body runs, which in this case happens five times and the increment operation which also happens five times. Thus just the for statement (excluding the body) has a total of 11 primitive operations.

While-do and repeat-until loops work in a similar fashion as shown below

| // Section 1abc <- 5while (abc >= 1) do abc-- // Section 2 abc <- 5 repeat  abc-- until abc = 1 |
| --- |

In the first section, the while-do loop, variable assignment happens right before it, a conditional check is done at the start of each body execution and the body runs indefinitely until that condition is met. In this case it is obvious there are 9 primitive operations under Section 1, and the same for Section 2, but oftentimes it's not possible to judge how long a while-do or repeat-until loop is going to run. In these cases, we use worst-case, best-case and average-case to determine the number of primitive operations.

We can also count primitive operations based on some input size of an argument to an algorithm. Such an example is shown below, where instead of a constant number of primitive operations, the number is related to the input size, more specifically as a function on the input size.

| Algorithm getFirstEven(A, n) Inputs: A the array to check  n the size of the array Output: the index of the first even number found in the array  for i <- 0 to n-1 do // 1 for assignment and n for condition if (A[i] % 2 = 0) then // 3 for indexing, modulo & comparison return A[i] // 1 for indexing i++ // 1 for incrementing return -1 |
| --- |

As it can be seen the for-to-do loops over n times, with one assignment operation at the start and *n* conditional checks. There are a total of 4 internal primitive operations , and 1 for the return statement at the end, thus we can count the total number of primitive operations as a function of the input n, such that f(n) = 4n + 1.

## 1.3 Asymptotic Analysis

When counting primitive operations, we discussed that it's often not possible to determine the exact number of primitive operations in some code due to control flows like while-do and repeat-until loops that can potentially run indefinitely. So we need a better way to describe an algorithm’s performance. One way is through asymptotic analysis.

Asymptotic analysis can help us describe the performance of an algorithm mathematically based on some based on its run-time performance, which can be described as a function of the input size of the arguments(s) to the algorithm. But before we can do so, we need some way of describing running time - the time taken by an algorithm to execute.

If we have the number of primitive operations expressed as a function of the input size as seen in the previous section, and assuming that each operation takes the same amount of time to run, then we can describe the running by using the function *T*. In the previous section, where f(n) = 4n + 1, the running time can be described as bounded by , for some constants *a* and *b* - that is, the running time is linear. Notice how the contant 1 was dropped. That is due to the fact we refer to a function's most fundamental or basic form, the largest degree term in polynomials, or the fastest growing term/function in an expression. The constant 1 does not change the fact the function is linear and by not having it there, does not make the function non-linear.

### § Big-O Notation

When describing the worst-case running time of a particular algorithm, we need a way of doing so. Big-O Notation is such a method. The goal is to find a mathematical upper bound for the running time or space complexity of a particular algorithm (i.e. its performance).

We introduce this notation with the following example.

| Algorithm isAllBigger(A, B, n) Inputs: A the first array to check B the second array to check n the size of the arrays Output: true if every element in A is bigger than B false otherwise  for i <- 0 to n-1 do // 1 for assignment and n for condition  for j <- 0 to n-1 do  if (A[i] <= B[j]) then // 3 for indexings & comparison  return false  j++ // 1 for incrementing i++ // 1 for incrementing return true |
| --- |

Counting the primitive operations in the above example. The outer for-to-do loop runs *n* times with 1 assignment operation before. This inner loop runs *n* number of times as well, with one 1 assignment operation and 1 increment operation after the loop. The inner body of the inner loop has 4 primitive operations. Thus, the total number of primitive operations for this algorithm can be calculated as follows, where *N* is the function for the number of primitive operations:

|  |  |
| --- | --- |

Thus the total number of primitive operations can be modelled by the polynomial function . When describing the running time *T* of an algorithm, it was shown that we can define a certain bounding for it. Big-O notation describes this as the performance of the function, specifically the worst-case performance of an algorithm. In the case of running time we said that constants and lower-order terms do not matter as the fastest growing term or function is considered. Thus *T(n)* in the above case would be bounded by . In Big-O notation, we say that the algorithm runs in .

In Big-O Notation the upper bound of a function is if for all .

This simply means that there is another function that is greater than or equal to for any point that is greater than a certain -value. Let’s prove that the function runs in .

| *Given , prove that run in , where the proposed is .*   |  |  | | --- | --- |   The quadratic formula can be applied here:   |  |  | | --- | --- |   Let :   |  |  | | --- | --- |   Using , due to the nature of a polynomial, for any , where and . Thus, runs in .  ***Visual Proof*** |
| --- | --- | --- | --- | --- | --- | --- |

### § Big-Omega Notation

Big-Omega Notation is similar to Big-O Notation, but instead it deems to find the mathematical lower-bound of an algorithm’s performance. It does so in the exact same process used in the previous section, however this time we aim to determine, from a given a and such that is if for all .

### § Big-Theta Notation

Big-Theta Notation aims to find out the tight-bound of a function (if it exists). If an and have been determined for a function, then we say that the function has a tight-bound if . In simpler terms, if the lower-bounds and upper-bounds are the same for a particular function, then we can say a tight-bound exists with the same as for both.

Mathematically, this is described as the following.

X`

is if there exist positive constants , , and such that for all .

### § Simple Method

Oftentimes, a proof can be quite difficult to do in order to determine the runtime of an algorithm with Big-O Notation. Thus we can apply certain rules to help us simplify an expression and get the Big-O form for the performance.

***Rule 1: Finds the fastest growing term***

This involves derivatives to determine the fastest growing term or by simply understanding the general nature of functions. It is easy and provable that an exponential function is a faster growing function than, let’s say, a logarithmic function. Thus, finding the fastest growing term can help us eliminate the other terms which do not contribute as much to the runtime of a particular algorithm. Let’s look at an example.

It is quite intuitive that is a much faster growing term/function than . We can use derivatives to prove this, but that would be beyond the scope of this course.

***Rule 2: Drop constant coefficients***

We have shown in previous examples that constants do not play much role in the long-term growth rate of the algorithm and thus they are insignificant and can be dropped. This is easily provable as in the proof example above.

***Rule 3: Drop additive constants***

Following *Rule 1*, constants like 1 and 500 can be dropped due to the fact that these do not change with increase in the input size.

|  | If anything has been missed, feel free to add what you feel needs to be explained to further improve the quality of the notes. |
| --- | --- |

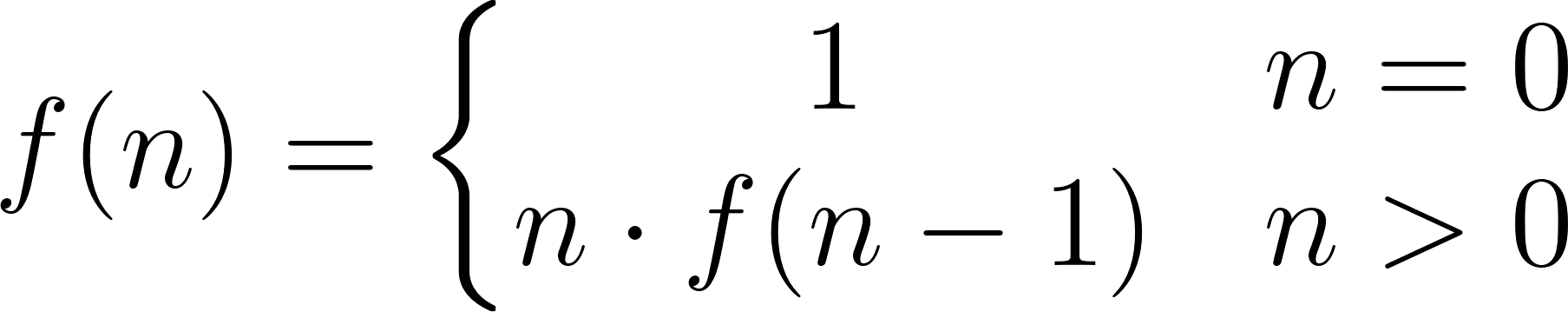
# 2 Recursive & Sorting Algorithms

| Terms & Definitions | |
| --- | --- |
| *Recursion* | In computer science, a function that invokes itself inside it, is called recursive. |
| *Base case* | Regarding a recursive function; the base case is the eventual stopping point at the end of every chain of recursive calls that are invoked by the function. |
| *Tail recursion* | Tail recursion is where recursion is called at the last step of the function. Requires the function to take in arguments that change in every invocation. |
| *Binary recursion* | Binary recursion is where two recursive calls are made in the same non-base case line |
| *Multiple recursion* | Multiple recursion is where *N* number of recursive calls are made in the same non-base case line. |

## 2.1 Mathematical Recursion

In mathematics, a recursively defined function is one that has a base case and is followed by some expression that calls or refers to itself. A popular recursively defined function, is one the factorial function.

Let be the -th factorial, then

[](https://www.codecogs.com/eqnedit.php?latex=f(n)%3D%5Cleft%5C%7B%5Cbegin%7Bmatrix%7D1%20%26%20n%3D0%5C%5Cn%5Ccdot%20f(n-1)%20%26%20n%3E0%5Cend%7Bmatrix%7D%5Cright.#0)

As you can see, recursive functions have two important features, a base case, and a recursive/inductive case. The base case is such that every valid input to the function will eventually invoke the base case - the base case is necessary to prevent infinite recursion. The recursive case makes use of its inputs, modifies it (if necessary), performs arithmetic calculations to them, or keeps track of them in some way such that another invocation to the function is a different one leading to the eventual base case. In the factorial example above, at every step, is taken and multiplied with the -th factorial.

## 2.2 Types of Recursion

In computer science, recursion is the same, in its principles but can be implemented in many ways. This section will discuss the different types of recursion, how they are implemented, and perform asymptotic analysis on popular recursive algorithms.

### § Linear Recursion

Let's begin by demonstrating a popular recursion function, the Colatz’s Conjecture with some pseudo-code.

| Algorithm colatzSteps(n) Input: n the starting number Output: the number of steps taken for the number to reach 1  if n = 1 then  return 0  else if (n % 2 = 0) then  return (1 + colatzSteps(n / 2))  else  return (1 + colazSteps(3 \* n + 1)) |
| --- |

A simple linear recursive function, but it has yet to be proven whether every positive integer has a finite number of steps following the above process i.e., will the base case always be reached for every positive integer?

The algorithm above can be described as linear recursive since every non-base step performs a single recursive call.

### § Tail Recursion

Tail recursion is another type of recursion, which often deals with multiple input parameters that change at every recursive call. In tail recursion, the recursive step contains a call to itself and nothing else. Understanding that state does not get preserved, such that variables defined before a recursive call get thrown out is critical. Tail recursion preserves the idea of state as it passes arguments (i.e. variables and other input parameters) to itself in the recursive step. Let's demonstrate how the classic factorial function can be written with tail recursion.

| Algorithm factorial(n, output) Inputs: n the number to determine the factorial of Output: the factorial of n  if n = 1 then  return output    factorial(n - 1, output \* n) |
| --- |

It is important to note that for this function to produce a valid answer, the output field will be initially set to one. The algorithm will invoke itself with a new output at each recursive step, until the base case is reached, then, it will return the output. It is important to note that any tail recursive function can be easily translated to an iterative function. The above example can be written with a while loop as shown below.

| Algorithm factorialIterative(n) Input: n the number to determine the factorial of  output <- 1  while n > 1 do  output <- output \* n  n--  return output |
| --- |

### § Binary & Multiple Recursion

Binary recursive algorithms have two calls to function in every non-base case (recursive step). An example is shown below

| Algorithm binarySum(A, start, end) Inputs: A the array to sum its elements  start the starting index to sum from  end the ending index to sum to  if (start = end) then  return A[start]  return binarySum(A, start, ⌊end / 2⌋) + binarySun(⌈end / 2⌉, end) |
| --- |

***NOTE: ⌊⌋, ⌈⌉ are the mathematical floor and ceil functions, respectively***

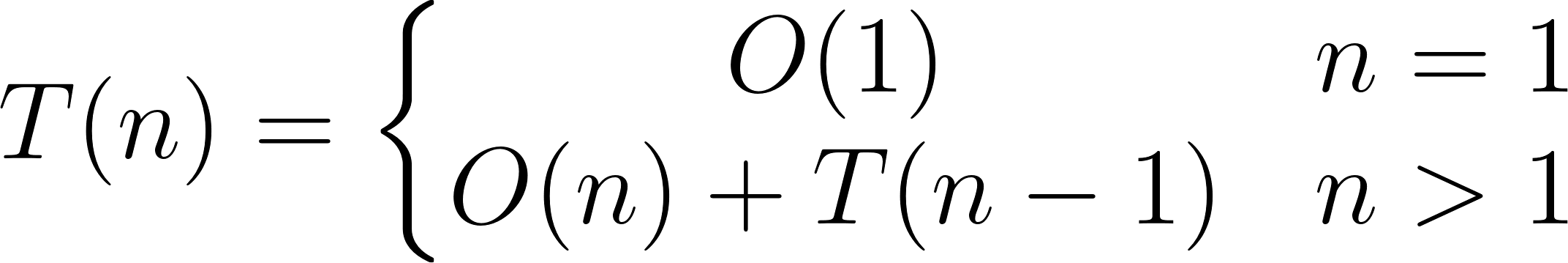
As you can see binary recursion makes use of invoking itself twice in the same recursive step with a certain relation between the two calls. Multiple recursion generalises binary recursion in that *n* calls are made to the function in each non-base case scenario.

## 2.2 Analysing Recursive Algorithms

Let’s take a look at a recursive sorting algorithm called selection sort. The algorithm is detailed below.

| Algorithm selectionSort(A, n) Inputs: A the array to be sorted in ascending order  n the length of the array  // post-condition: A will be sorted  if n = 1 then  return  // determining the maximum value’s index between 0 and n  maxValueIndex <- 0  for i <- 0 to n - 1 do  if (A[i] >= A[maxValueIndex]) then // 3 primitive operations  maxValueIndex <- i  // swapping last value with the max value in the array  temp <- A[n-1]  A[n-1] <- A[maxValueIndex]  A[maxValueIndex] <- temp  // calling selection sort on the first n-1 elements of the array  selectionSort(A, n-1) |
| --- |

To determine the runtime complexity of the algorithm, we need to first count the primitive operations in the non-recursive cases and then define recursively. It can be seen that the algorithm first loops through the array from element 0 to *n-1*, and determines the largest value found so far and keeps track of its index. There is one primitive operation in the checking of the base case, one to initialise the variable storing the maximum value’s index. The for-to-do loop runs *n* times, with 1 conditional check, 3 primitive operations; two indexing and one comparison, plus one more for changing value. The swapping process counts as 8 primitive operations. Overall, we can say that:

[](https://www.codecogs.com/eqnedit.php?latex=T(n)%3D%5Cleft%5C%7B%5Cbegin%7Bmatrix%7D%20O(1)%20%26%20n%3D1%5C%5CO(n)%2BT(n-1)%20%26%20n%3E1%20%5Cend%7Bmatrix%7D%5Cright.#0)

We can evaluate this in the following way

|  |  |
| --- | --- |

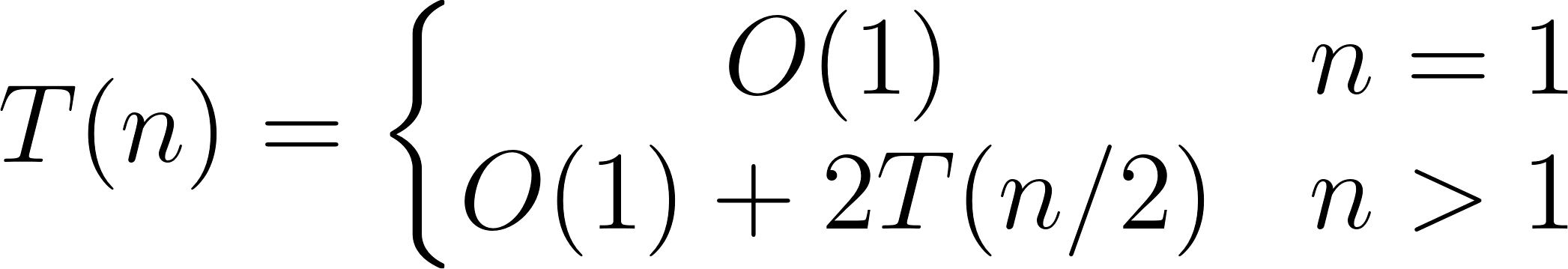
Since *n* decreases by 1 at each recursive call, there will be *n* recursive calls in total. Thus the total number of primitive operations *N* can be calculated as follows.

Thus the algorithm runs in .

Let’s go back and analyse the runtime complexity of the binary sum algorithm.

| Algorithm binarySum(A, start, end) Inputs: A the array to sum its elements  start the starting index to sum from  end the ending index to sum to  if (start = end) then  return A[start]  return binarySum(A, start, ⌊end / 2⌋) + binarySun(⌈end / 2⌉, end) |
| --- |

***NOTE: ⌊⌋, ⌈⌉ are the mathematical floor and ceil functions, respectively***

[](https://www.codecogs.com/eqnedit.php?latex=T(n)%3D%5Cleft%5C%7B%5Cbegin%7Bmatrix%7D%20O(1)%20%26%20n%3D1%5C%5CO(1)%2B2T(n%2F2)%20%26%20n%3E1%20%5Cend%7Bmatrix%7D%5Cright.#0)

|  | .  .    =2^log (base 2) n  =n  Is it O(n)  **T(n) = 2T(n-1)+1**  T(n) = 2[2T(n-2)+1]+1  T(n) = 2^2[2T(n-3)+1]+2+1  T(n) = 2^3T(n-3)+2^2+2+1  .  .  T(n) = 2^k(T(n-k) + 2^(k-1) + 2^(k-2) + … + 4 + 2 + 1    Assume n-k=0  n=k    =2^nT(0)+1+2+2^2+….+2^k-1  =2^n x 1 + 2^k -1  = 2^n + 2^n -1  = 2^(n+1) -1  O(2^n) |
| --- | --- |

### <========================{ WORK IN PROGRESS }========================>

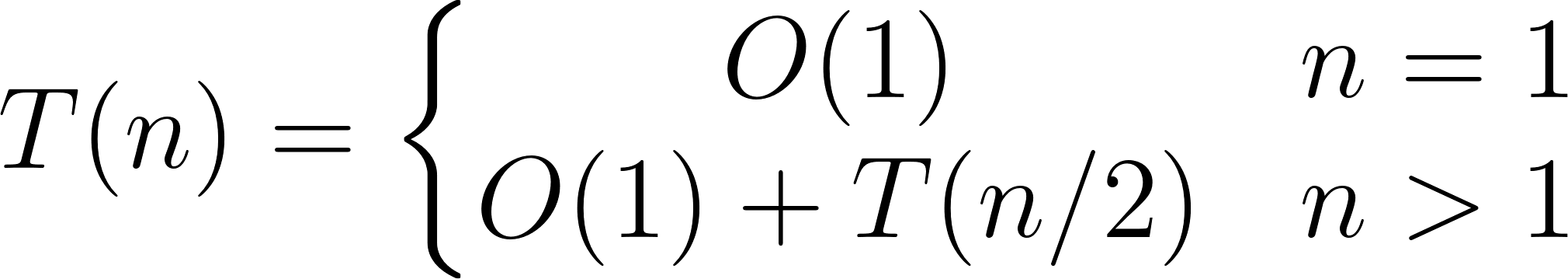
## 2.3 Sorting Algorithms

### § Divide-and-Conquer Paradigm

The Divide-and-conquer paradigm makes use of splitting the input size into smaller parts. and invoking each part in a call to itself. The calls can either have some relation with one or another or can be isolated into separated recursive cases, like the binary search algorithm displayed below.

| Algorithm binarySearch(A, start, end, k) Inputs: A a sorted array to search into  start the starting index to search from  end the ending index to search to  k the element to search for  if (start < end) then  return false  middle = ⌊(start + end) / 2⌋  if (k = A[middle]) then  return true  else if (k < A[middle]) then  binarySearch(A, start, middle-1)  else   binarySearch(A, middle+1, end) |
| --- |

Analysing the binary search algorithm requires determining as a recursively defined function.

[](https://www.codecogs.com/eqnedit.php?latex=T(n)%3D%5Cleft%5C%7B%5Cbegin%7Bmatrix%7D%20O(1)%20%26%20n%3D1%5C%5CO(1)%2BT(n%2F2)%20%26%20n%3E1%20%5Cend%7Bmatrix%7D%5Cright.#0)

|  |  |
| --- | --- |

The total number of calls to the function, the function call stack, will have a maximum height as only half of the provided input size is considered at each recursive call. Thus the algorithm runs in time - that is, there will be a maximum of Big-O of 1s expressed in .

### § Merge-Sort

Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm. The following is the pseudo-code for the merge-sort algorithm.

| Algorithm mergeSort(A, start, end) Inputs: A an array to sort  start the starting index to sort from  end the ending index to sort to  if (start > end) then  return middle = ⌊(start + end) / 2⌋  mergeSort(A, start, middle) // sort the left half  mergeSort(A, middle+1, end) // sort the right half  merge(A, start, middle, end) // merge the two halves  Algorithm merge(A, start, middle, end) Inputs: A an array of two sorted halves A[start:middle] and  A[middle,end]  start the starting index of the left half  middle the middle index dividing the sorted halves  end the ending index of the right half  Output: A sorted A[start:end]  // get sizes of the two parts and store a copy of each  leftSize = middle - start + 1  rightSize = end - start  leftArray = copy(A, start, middle)  rigthArray = copy(A, middle, end)  // initialise the pointer variables  leftPointer = 0  rightPointer = 0  arrayPointer = l  // compare and sort accordingly  while (left < leftSize) and (right < rightSize) do  if leftArray[leftPointer] <= rightArray[rightPointer] then  A[arrayPointer++] <- leftArray[leftPointer++]  else  A[arrayPointer++] <- rightArray[rightPointer++]  // append the leftovers in the left array  while (leftPointer < leftSize) do  A[arrayPointer++] <- leftArray[leftPointer++]  // or alternatively, append the leftovers in the right array  while (rightPointer < rightSize) do  A[arrayPointer++] <- rightArray[rightPointer++] |
| --- |

Merge-sort works in three stages. The *Divide*, *Recur* and *Conquer* steps. Below is a visualisation of merge-sort showing all three steps.

***Divide:***

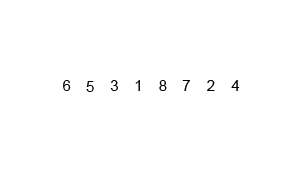
Partitions the array into two halves. This splits off the array to deal with sorting each half.

***Recur:***

Merge-sort is applied to each half. This is the recursive step.

***Conquer:***

Each sorted half, at every step, is joined together to get a union sorted list.



*Figure 2: Animation of Merge-sort*

### <========================{ WORK IN PROGRESS }========================>

### § Quick-Sort

Another popular sorting algorithm is quick-sort, which, much like merge-sort, utilises the divide-and-conquer paradigm to sort. However, one key difference is that quick-sort utilises randomised pivots instead of dividing into equal halves. The algorithm is described below.

| Algorithm quickSort(A) Inputs: A an array to sort  if length(A) <= 1 then  return A  pivot = randomInt(0, length(A))  lesser, equal, greater <- partition(A, pivot)  quickSort(lesser)  quicksort(greater)  return join(lesser, equal, greater)   Algorithm partition(A, pivot) Inputs: A an array to be partitioned  pivot the pivot to partition around   Output: lesser, equal and greater arrays  // initialise empty arrays  lesser = []  equal = []  greater = []  pivot = A[pivot]  for i <- 0 to length(A) do  element = A[i]  if (element = pivot) then  equal.append(element)  else if (element < pivot) then  lesser.append(element)  else  greater.append(element)  return (lesser, equal, greater) Algorithm join(lesser, equal, greater)  Inputs: lesser the array to join before equal  equal the middle array  greater the array to join after equal   Output: a joined array in the order, lesser, equal, greater  A = []  while not lesser.isEmpty() do  A.append(lesser.get(lesser.first()))  while not equal.isEmpty() do  A.append(equal.get(equal.first()))  while not greater.isEmpty() do  A.append(greater.get(greater.first()))  return A |
| --- |

The runtime complexity of quick-sort can be observed as

### § In-Place Quick-Sort

# 3 Advanced Sorting Algorithms

## 3.1 Bucket-Sort

## 3.2 Lexicographic Sort

## 3.2 Radix-Sort

## 3.2.1 Radix-Sort for Binary

# 4 Linear Data Structures

## 4.1 Arrays

## 4.2 Linked Lists

### § Singly Linked List

### § Doubly Linked List

### § Circularly Linked List

## 4.3 Stacks

## 4.4 Queues

## 4.5 Applications of Stacks & Queues

# 5 Lists & Iterators

## 5.1 Array Lists

## 5.2 Amortisation

## 5.3 Positional Lists

## 5.4 Iterators

# 5 Introduction to Trees

## 5.1 Tree Basics

## 5.2 Tree Traversals

## 5.3 Binary Trees

# 6 Priority Queues, Maps & Sets

## 6.1 Priority Queues

## 6.2 Maps

## 6.3 Sets

# 7 Hash Tables & Binary Search Trees

## 7.1 Hash Tables

## 7.2 Binary Search Trees

# 8 Search Trees

## 8.1 Splay Trees

## 8.2 (2,4) Trees

### § Red-Black Trees

### § B-Trees

# 9 Introduction to Graphs

## 9.1 Graph Fundamentals

## 9.2 Graph Implementations

## 9.3 Depth-First Search

## 9.4 Breadth-First Search

# 10 More Graph Stuff

## 10.1 Shortest-Path Algorithms

## 10.2 Minimum Spanning Trees

# 11 Text Processing

## 11.1 Strings & Pattern Matching

## 11.2 Tries

## 11.3 Text Compression

# 12 Advanced Topics in ADS

## 12.1 Skip List

## 12.2 Selection Problem

## 12.3 PQ SOrting

### **<**========================**{** **WORK IN PROGRESS }========================>**