**Programming Notations and Notions:**

Errors in Programmin:

1) Syntax errors:

Syntax refers to the structure of a program and the rules about that structure.

2) Runtime errors:

The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened.

3) Semantic Errors:

The third type of error is the  If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages. However, your program will not do the right thing. It will do something else. .

**Algorithms:**

An algorithms for a paritcular task can be defined as “a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time”. As such, an algorithm must be precise enough to be understood by human beings.

Recursion:

Recursion in a nutshell is to solve a problem by making subproblems of that problem then solving sub problems then eventually solving the original problem using that sub problem.

Recursion is implemented by using functions. Each time a function is called using recursive approach the function copies itself but smaller to the original one (this is done by calling the function in its own definition but with smaller base case) and this continues untill we get a base case (situation where we cannot make subproblems) then we solve the lowest subproblem then using its solution we solve its parent problem then eventually we solve the original problems

Rules of Recursion:

1) A recursive algorithm must have a base case.

2) A recursive algorithm must change its state and move toward the base case.

3) A recursive algorithm must call itself, recursively.

Monotonicity:

The monotonicity of a function gives an idea about the behaviour of the function. A function which is either completely non-increasing or completely non-decreasing is said to be monotonic.

A function is said to be monotonic if it is either increasing or decreasing in its entire domain.

eg : f(x) = 2x + 3 is an increasing function while f(x) = -x3 is a decreasing function.

Floor and Ceiling:

The floor and ceiling functions give us the nearest integer up or down.

Example: floor(2.3) = 2

ceiling(2.3) = 3

Modular Arithmetic:

Sometimes, we are only interested in what the **remainder** is when we divide *A* by *B*.  
For these cases there is an operator called the modulo operator (abbreviated as mod).

A

B

Using the same *A*, *B*, *Q*, and *R* as above, we would have: *A* mod *B*=*R*

R

Q

B

A

We would say this as *A* *modulo* *B* *is equal to* *R*. Where *B* is referred to as the **modulus**.

A

B

R

B

For example:

13 mod 5 = 3

Infix, prefix and postfix operations:

When you write an arithmetic expression such as B \* C, the form of the expression provides you with information so that you can interpret it correctly. In this case we know that the variable B is being multiplied by the variable C since the multiplication operator \* appears between them in the expression. This type of notation is referred to as infix since the operator is in between the two operands that it is working on

What would happen if we moved the operator before the two operands? The resulting expression would be + A B. Likewise, we could move the operator to the end. We would get A B +. These look a bit strange.

These changes to the position of the operator with respect to the operands create two new expression formats, **prefix** and **postfix**. Prefix expression notation requires that all operators precede the two operands that they work on. Postfix, on the other hand, requires that its operators come after the corresponding operands. A few more examples should help to make this a bit clearer

A + B \* C would be written as + A \* B C in prefix. The multiplication operator comes immediately before the operands B and C, denoting that \* has precedence over +. The addition operator then appears before the A and the result of the multiplication.

In postfix, the expression would be A B C \* +. Again, the order of operations is preserved since the \* appears immediately after the B and the C, denoting that \* has precedence, with + coming after. Although the operators moved and now appear either before or after their respective operands, the order of the operands stayed exactly the same relative to one another.

1) Binary Search:

1. Let “min = 0” and “max = n-1”.
2. If “max < min”, then stop: “target is not present in array”. Return “-1”.
3. Compute “guess as the average of max and min”, rounded down (so that it is an integer).
4. If “array[guess] equals target”, then stop. You found it! Return “guess”.
5. If the “guess” was too low, that is, “array[guess] < target”, then set “min = guess + 1”.
6. Otherwise, the guess was too high. Set “max = guess - 1”.
7. Go back to step 2.

2) Selection Sort:

There are many different ways to sort the cards. Here's a simple one, called **selection sort**, possibly similar to how you sorted the cards above:

1. Find the smallest card. Swap it with the first card.
2. Find the second-smallest card. Swap it with the second card.
3. Find the third-smallest card. Swap it with the third card.
4. Repeat finding the next-smallest card, and swapping it into the correct position until the array is sorted.

3) Insertion Sort:

Now that you know how to insert a value into a sorted subarray, you can implement insertion sort:

1. Call insert to insert the element that starts at index 1 into the sorted subarray in index 0.
2. Call insert to insert the element that starts at index 2 into the sorted subarray in indices 0 through 1.
3. Call insert to insert the element that starts at index 3 into the sorted subarray in indices 0 through 2.
4. …
5. Finally, call insert to insert the element that starts at index *n*−1 into the sorted subarray in indices 0 through *n*−2.

Asymptotic Notations:

When it comes to analysing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations, also known as Asymptotic Notations.

The word Asymptotic means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

## Types of Asymptotic Notations

We use three types of asymptotic notations to represent the growth of any algorithm, as input increases:

1. Big Theta (Θ)
2. Big Oh(O)
3. Big Omega (Ω)

### Tight Bounds: Theta

When we say tight bounds, we mean that the time compexity represented by the Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be.

### Upper Bounds: Big-O

This notation is known as the upper bound of the algorithm, or a Worst Case of an algorithm.

It tells us that a certain function will never exceed a specified time for any value of input n.

Lower Bounds: Omega

Big Omega notation is used to define the **lower bound** of any algorithm or we can say **the best case** of any algorithm.

This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.

# Space Complexity of Algorithms

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

***Space complexity****is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.*

Sometime **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

**Space Complexity** = **Auxiliary Space + Input space**

## Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space**

It's the amount of memory used to save the compiled version of instructions.

1. **Environmental Stack**

Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.

For example, If a function A() calls function B() inside it, then all th variables of the function A() will get stored on the system stack temporarily, while the function B() is called and executed inside the funciton A().

1. **Data Space**

Amount of space used by the variables and constants.

For calculating the space complexity, we need to know the value of memory used by different type of datatype variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

|  |  |
| --- | --- |
| Type | Size |
| bool, char, unsigned char, signed char, \_\_int8 | 1 byte |
| \_\_int16, short, unsigned short, wchar\_t, \_\_wchar\_t | 2 bytes |
| float, \_\_int32, int, unsigned int, long, unsigned long | 4 bytes |
| double, \_\_int64, long double, long long | 8 bytes |

**Data Structures:**

The term data structure is used to denote a particular way of organizing data for particular types of operation. Often we want to talk about data structures without having to worry about all the implementational details associated with particular programming languages or how the data is stored in computer memory. We can do this by formulating abstract mathematical models of particular classes of data strucures or data types which have common features. These are called abstract data types, and are defined only by the operations that may be performed on

them. Typically, we specify how they are built out of more primitive data types (e.g., integers

or strings), how to extract that data from them, and some basic checks to control the flow of

processing in algorithms. The idea that the implementational details are hidden from the user

and protected from outside access is known as encapsulation.

1) Stacks:

A stack (sometimes called a “push-down stack”) is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the “top.” The end opposite the top is known as the “base.”

This ordering principle is sometimes called LIFO, last-in first-out. It provides an ordering based on length of time in the collection. Newer items are near the top, while older items are near the base.

The stack abstract data type is defined by the following structure and operations. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the “top.” Stacks are ordered LIFO. The stack operations are given below.

* Stack() creates a new stack that is empty. It needs no parameters and returns an empty stack.
* push(item) adds a new item to the top of the stack. It needs the item and returns nothing.
* pop() removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
* peek() returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
* isEmpty() tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
* size() returns the number of items on the stack. It needs no parameters and returns an integer.

2) Queue:

A queue is an ordered collectionof items where the addition of new items happens at one end, called the “rear,” and the remvoal of existing itms occurs at the other end, commonly called the “front.” As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.

The most recently added item in the queue must wait at the end of the collection. The item that has been in the collection the longest is at the front. This ordering principle is sometimes called FIFO, first-in-first-out it is also known as “first-come first-served.”

The queue abstract data type is defined by the following structure and operations. A queue is structured, as described above, as an ordered collection of items which are added at one end, called the “rear,” and removed from the other end, called the “front.” Queues maintain a FIFO ordering property. The queue operations are given below.

* Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.
* enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.
* dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
* isEmpty() tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
* size() returns the number of items in the queue. It needs no parameters and returns an integer.

3) Deque:

A deque, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure. [Figure 1](https://runestone.academy/runestone/books/published/pythonds/BasicDS/WhatIsaDeque.html" \l "fig-basicdeque) shows a deque of Python data objects.

The deque abstract data type is defined by the following structure and operations. A deque is structured, as described above, as an ordered collection of items where items are added and removed from either end, either front or rear. The deque operations are given below.

* Deque() creates a new deque that is empty. It needs no parameters and returns an empty deque.
* addFront(item) adds a new item to the front of the deque. It needs the item and returns nothing.
* addRear(item) adds a new item to the rear of the deque. It needs the item and returns nothing.
* removeFront() removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
* removeRear() removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
* isEmpty() tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
* size() returns the number of items in the deque. It needs no parameters and returns an integer.

4) Unordered Lists:

Throughout the discussion of basic data structures, we have used Python lists to implement the abstract data types presented. The list is a powerful, yet simple, collection mechanism that provides the programmer with a wide variety of operations. However, not all programming languages include a list collection. In these cases, the notion of a list must be implemented by the programmer.

A List is a collectionof items where each item holds a relative position with respect to the others. More specifically, we will refer to this type of list as an unordered list.

This means that each memory block which contains element of the list must contain address of next memory block as well as having the element of the list itself and the last memory block must contain element along with an empty address which points to nothing or None (in python).

The structure of an unordered list, as described above, is a collection of items where each item holds a relative position with respect to the others. Some possible unordered list operations are given below.

* List() creates a new list that is empty. It needs no parameters and returns an empty list.
* add(item) adds a new item to the list. It needs the item and returns nothing. Assume the item is not already in the list.
* remove(item) removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
* search(item) searches for the item in the list. It needs the item and returns a boolean value.
* isEmpty() tests to see whether the list is empty. It needs no parameters and returns a boolean value.
* size() returns the number of items in the list. It needs no parameters and returns an integer.
* append(item) adds a new item to the end of the list making it the last item in the collection. It needs the item and returns nothing. Assume the item is not already in the list.
* index(item) returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
* insert(pos,item) adds a new item to the list at position pos. It needs the item and returns nothing. Assume the item is not already in the list and there are enough existing items to have position pos.
* pop() removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
* pop(pos) removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

5) Ordered Lists:

We will now consider a type of list known as an ordered list. For example, if the list of integers shown above were an ordered list (ascending order), then it could be written as 17, 26, 31, 54, 77, and 93. Since 17 is the smallest item, it occupies the first position in the list. Likewise, since 93 is the largest, it occupies the last position.

The structure of an ordered list is a collection of items where each item holds a relative position that is based upon some underlying characteristic of the item. The ordering is typically either ascending or descending and we assume that list items have a meaningful comparison operation that is already defined. Many of the ordered list operations are the same as those of the unordered list.

* OrderedList() creates a new ordered list that is empty. It needs no parameters and returns an empty list.
* add(item) adds a new item to the list making sure that the order is preserved. It needs the item and returns nothing. Assume the item is not already in the list.
* remove(item) removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
* search(item) searches for the item in the list. It needs the item and returns a boolean value.
* isEmpty() tests to see whether the list is empty. It needs no parameters and returns a boolean value.
* size() returns the number of items in the list. It needs no parameters and returns an integer.
* index(item) returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
* pop() removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
* pop(pos) removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.