Scala

- Basic

  Types: Char 16-bit unsigned integer Byte 8-bit signed integer

  Short 16-bit signed integerInt32-bit signed integer

  Long 64-bit signed integer Float single-precision floating-point number

  Double double-precision floating-point number

  Unit no value – declared by ()

  Each expression has a type.

- Class

  - Generic classes take a type as a parameter within square brackets []. One convention is to use the letter A as type parameter identifier, though any parameter name may be used.

  ```
  class Generic[A] {
      var member = 0          ← Members Variable
      def Function (x: A) {……}
      }
  ```

  -class normal OOP class.

  -object similar to class but only one instance may exist.

  -trait also similar, but cannot be instantiated.

  -case class similar to class, provides special functionality.

- Call stack
  - Function call will add (push) to the top of the stack or create a stack frame.
  - Most recent called function is on the top of the stack, active frame.
  - When function finishes its work, it frame is popped off of the stack, the frame immediately below it becomes the new, active, function on the top of the stack.

- References
  - Reference is a value that enables a program to indirectly access a particular datum, such as a variable's value or a record, in the computer's memory or in some other storage device.
  - The reference is said to refer to the datum, and accessing the datum is called dereferencing the reference.

- Tail recursion
  - Perform the calculations first, passing the results of the current step to the next recursive step.
  - Tail recursion returns the result at the last step of this recursion call.

Runtime

- O, Ω, Θ, Amortized (average run time)

  g(n)

  f(n) = O(g(n)) —— f(n) is bounded above by a constant scaling of g(n) for large n

  f(n) grows no faster than g(n).

  f(n) = Ω(g(n)) —— f(n) is bounded below by a constant scaling of g(n) for large n

  f(n) grows no slower than g(n).

  f(n) = Θ(g(n)) —— f(n) is bounded above and below by a constant scaling of g(n) for large n. f(n) runs the same as g(n).

- Deriving time/space usage from code

- Deriving a recurrence for time/space usage from code.

  Master Method

  $T(n) = a\,T(n/b) + f(n)$ where $a \geq 1$ and $b > 1$

  $f(n) = \Theta(n^c)$

  1. $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$
  2. $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$
  3. $c > \log_b a$ then $T(n) = \Theta(n^c)$

logarithms example:

$\log_2(8) = 3$ $2^3 = 8$

$\log_5(625) = 4$ $5^4 = 625$

- Summations

R1. $\sum_{i=j}^{k}(cf(i)) = c\sum_{i=j}^{k} f(i)$, for $j \leq k$.

R2. $\sum_{i=j}^{k}(f(i) + c) = \sum_{i=j}^{k} f(i) + \sum_{i=j}^{k} c$, for $j \leq k$.

R3. $\sum_{i=j}^{k} c = (k - j + 1)c$, for $j \leq k$.

R4. $\sum_{i=j}^{k} f(i) = \sum_{i=\ell}^{k} f(i) - \sum_{i=\ell}^{j-1} f(i)$, for $j > \ell$.

R5. $\sum_{i=j}^{k} f(i) = f(j)+f(j+1)\ldots+f(k-1)+f(k)$, for $j \leq k$.

R6. $\sum_{i=j}^{k} f(i) = f(j) + \ldots + f(\ell - 1) + \sum_{i=\ell}^{k} f(i)$, for $j < \ell \leq k$.

R7. $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$.

R8. $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$.

R9. Sterling: $n! \leq c_s n^n$ is a tight upper-bound (some constant $c_s > 0$ exists).

O (1) time

- Accessing Array Index (int a = ARR [5] ;)
- Inserting a node in Linked List
- Pushing and Popping on Stack
- Insertion and Removal from Queue
- Finding out the parent or left/right child of a node in a tree stored in Array
- Jumping to Next/Previous element in Doubly Linked List

O (n) time

In a nutshell, all Brute Force Algorithms, or Noob ones which require linearity, are based on O(n) time complexity

- Traversing an array
- Traversing a linked list
- Linear Search
- Deletion of a specific element in a Linked List (Not sorted)
- Comparing two strings
- Checking for Palindrome
- Counting/Bucket Sort and here too you can find a million more such examples....

O (log n) time

- Binary Search
- Finding largest/smallest number in a binary search tree
- Certain Divide and Conquer Algorithms based on Linear functionality
- Calculating Fibonacci Numbers - Best Method The basic premise here is NOT using the complete data, and reducing the problem size with every iteration

O (n log n) time

The factor of 'log n' is introduced by bringing into consideration Divide and Conquer. Some of these algorithms are the best optimized ones and used frequently.

- Merge Sort
- Heap Sort
- Quick Sort
- Certain Divide and Conquer Algorithms based on optimizing O(n^2) algorithms

O (n^2) time

These ones are supposed to be the less efficient algorithms if their O(n log(n)) counterparts are present. The general application may be Brute Force here.

- Bubble Sort
- Insertion Sort
- Selection Sort
- Traversing a simple 2D array

Containers / Container Operations

- mutable.Seq (apply, update, length, iterator)
- Iterator
- ListADT and concrete classes
- Singly-Linked/Doubly-Linked List – SNode vs DNode
- Positional Linked List – PA1
- Sorted ListADT and concrete classes (insert, remove, apply, find)

Sequences, Arrays, Lists

ADT: Abstract data type

seq/list

get(i)   set(i, elem)   insert(i, elem)   remove(i, element)

replace elem   insert elem at i   remove i ~~and~~ with a new value.

mutable seq ADT:

apply(i)   iterator   length   update(i)

used once   replace i with new value.

Array: ⊕ random access (fast), access as seq | for array:

⊖ Fixed size, Empty values/entries | $O(1)$: apply, length, update.

positional Insert and Remove: $O(n)$ — worst case.

insert: - - ↑ - - → - - * - - from (position to end-1) shift to Right. by 1.

remove: - - ⊖ - - → - - - - from (position +1 to end-1) shift to ~~rig~~ left by 1.

Reserve storage: $\theta(n)$ occurs when size is 0 or 2^k

Insert 'next' to end: $\boxed{O(1)}$

worst case: if storage is full, reserve space, runtime = $O(n^2)$

Insert n items to the end:

- n operations: each $O(1)$, total $= \sum_{i=1}^{n} O(1) = \boxed{\theta(n)}$
- Double costs: when $i = 2^k$, $k=0$ or $k = Last - 1$

  $n = 2^k + 1$, $k = \log(n-1)$
- reverse log(n) space: $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

  since $k = \log(n-1)$, $2n-3$

Total: $T(n)$ = insertion + reverse $= \theta(n) + \theta(n) = \theta(n)$

Function Analysis:

Log Refresher: $\boxed{\text{base } 2}$

$\log(n^a) = a\log(n)$

$\log(an) = \log(a) + \log(n)$

$\log(\frac{a}{n}) = \log(a) - \log(n)$

change base:

b to c: $\log_b(n) = \dfrac{\log_c(n)}{\log_c(b)}$

$b^{\log_b(n)} = \log_b(b^n) = n$

f(n)

$\log(n)$

n

$n\log(n)$

$n^{\#}$

2^n

$n!$

grows

- $f(n) = \theta(g(n))$:

---- bound above and below ----

— f(n) runs the same as g(n)

Common Runtimes:

$\theta(1)$: constant time $\quad T(n) = \theta(1)$

$\theta(\log(n))$: logarithmic time

$\theta(n)$: linear time

$\theta(n^2)$: quadratic time

$\theta(n^k)$: polynomial time

$\theta(n^c)$: exponential time

Big-O: upper bound

Big-Ω: lower bound

Big θ: upper & lower bound

if g(n)

- $f(n) = O(g(n))$:

f(n) is bound above by const. scaling of g(n)

Grows no faster than g(n)

- $f(n) = \Omega(g(n))$:

.... bound below by const. scaling ....

Grows no slower than g(n)

loop: $O(n)$ $\quad$ loops: $O(n^{\#})$

if: $O(1)$

Random access: $O(1)$

linear access: $O(n)$

List Runtime:

Function	Unsorted	Positional
Apply	Θ(i)	Θ(i)
Update	Θ(i)	O (1)
Insert	Θ(i)	O (1)
remove	Θ(i)	O (1)

ArrayList iterator runtime: Θ(i) $\quad$ Access faster

Linked List iterator runtime: Θ(i) $\quad$ Access slower

	ArrayList			Linked List	
Function	Unsorted	Sorted		Unsorted	Sorted
Apply	O (1)	O (1)		O(i)	O(i)
Update	O(i)	-		O (1) *	-
Insert	O(n)	O(n)		O (1) *	O(n)
Remove	O(n)	O(n)		O (1) *	O(n)
Insert (n, e)	A O (1)	-		O (1)	-

A: Amortized, average runtime. $\quad$ *: position operation runtimes

Search / Finding Elements / Data Access

- Data Access

  – Random Access (e.g., unsorted Array List)

  Direct access, no matter how many elements in a set (array, seq) runtime is O (1).

  – Restricted Access - ordered insert/only front element/etc. (Stack, sorted ArrayList, etc.)

- Iterators

  – How to iterate through a sequence (starting and terminating conditions)

  HasNext() is true, then Next() is executable. Move to next position.

  – How to operate on a sequence (iterator/position)

- Linear vs Binary search

  Linear – from first index all the way though the end and search for the element

  Binary – Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Runtime (linear vs binary search)

	Linear	Binary
Best	O (1)	O (1)
Average	O (n)	log (n)
Worst	O (n)	log (n)

Data Organization

- Linear, contiguous
- Linear, linked

Sorting

- Merge Sort:

  Split the original array/ seq/ list into to two sub sets by the middle index of array. Do this for the sub sets of the sub sets until there is only one element left in a subset.
  Compare the value of subsets and make arrangement, then recombine them into one.

- Quick Sort:

  Select a pivot, move every thing less then it to left, larger then it to the right.

  Select a pivot for the left (index 0 till the index of the first step pivot)

  Select a pivot for the right (index of the first step pivot till the length-1)

  Do this for many times in recursion, until there is only one element in

  the n^{th} left or right.

- Bubble sort

- Insertion sort

Runtime

<table>
<tr><th></th><th>Merge Sort</th><th>Quick Sort</th><th>Bubble Sort</th><th>Insertion Sort</th></tr>
<tr><td>Best</td><td rowspan="3">n log(n)</td><td>n log(n)</td><td rowspan="3">n^2</td><td rowspan="3">n^2</td></tr>
<tr><td>Average</td><td>n log(n)</td></tr>
<tr><td>Worst</td><td>n^2</td></tr>
</table>