# Introduction to Data Structures and Algorithms

Effective:

1. Time Complexity
2. Space Complexity

## Best/Worst/Average Case

1. **Worst Case is usually used**. It is an **upper bound** and in certain application domains(Eg: Air Traffic control, Surgery) knowing the worst case time complexity is of crucial importance.
2. For some algorithms, worst case occurs fairly often.
3. Average Case is often as bad as the worst case.
4. Finding average case can be very difficult.

## Asymptotic Analysis

**Goal:** To simplify analysis of running time by getting rid of details which may be affected by specific implementation and hardware.

Example: rounding 1, 000,001 to 1, 000, 000

3n2 = n2

**Capturing the essence:**

How the running time of an algorithm increases with the size of the input.

## Asymptotic Notation

1. The “big-Oh” O-notation
2. Asymptotic upper bound
3. Used for worst case analysis.

**Simple Rule**:

Drop lower order terms and constant factors.

50nlogn = O(nlogn)

7n – 3 = O(n)

8n2logn + 5n2 + n = O(n2logn)

Even though 50nlogn is O(n5), it is expected that such an approximation be of as small an order as possible.

## Asymptotic Analysis of Running Time

1. Use O-notation to express number of primitive operations executed as function of input size.
2. Comparing asymptotic running times

* An algorithm that runs in O(n) time is better than one that runs in O(n2) time.
* Similarly, O(logn) is better than O(n)
* **Logn < n < n^2 < n^3 < 2^n**

1. Caution: Beware of very large constant factors.

An algorithm running in time 1,000,000 n is still O(n) but might be less efficient than one running in time 2n^2 which is O(n2).

## Asymptotic Notation

Special classes of algorithms:

1. **Logarithmic**: O(logn)
2. **Linear**: O(n)
3. **Quadratic**: O(n2)
4. **Polynomial**: O(n^k), k >=1
5. **Exponential**: O(a^n), a > 1

## Relatives of the Big-Oh

1. Ω(f(n)): Big Omega – Asymptotic lower bound
2. O(f(n)): Big Theta – Asymptotic tight bound
3. Little-Oh
4. Little-omega

## Analogy with real numbers

F(n) = O(g(n)) f <= g – Big Oh

F(n) = Ω(g(n)) f >= g – Big Omega

F(n) = theta(g(n)) f = g – Big Theta

F(n) = o(g(n)) f < g – Little Oh

F(n) = ꙍ(g(n)) f > g – Little Omega

# 2: Stacks

## Abstract Data Types (ADTs)

What is Abstract Data Type?

Abstract Data Type defines a set of it’s instances with:

1. **Specific Interface** – a collection of signatures of operations that can be invoked on an instance.
2. **Set of Axioms** – (Precondition and Postcondition) that defines the semantics of operations (What operations instances of ADT do, but not how)

## Types of Operations:

1. Constructors – Create instance of that particular data type – New()
2. Access functions – Access elements of data type - IsIn()
3. Manipulation Procedures – Modify the elements of that data type. – insert() and delete()

## Simple ADTs

1. Queue
2. Deque
3. Stack

## Stack

1. A stack is a container of objects that are inserted and removed according to the Last-In-First-Out(**LIFO**) principle.
2. Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
3. Inserting an item is known as “**Pushing**”

Removing an item is known as “**Popping**”

## Stack Methods

1. Empty() – Tests if this stack is empty
2. Peek() – Looks at the object at the top of this stack without removing it from the stack. Throws EmptyStackException if the stack is empty.
3. Pop() – Removes the object at the top of this stack and returns that object as the value of this function. Throws EmptyStackException if the stack is empty.
4. Push() – pushes an item on to the top of this stack
5. Search(Object o) – Returns the 1-based position where an object is on this stack.

The topmost item on the stack is considered to be at distance 1.

## Methods inherited from java.util.Vector

Size

isEmpty

addAll

clear

iterator

removeAll

remove

add etc

## Interfaces:

1. Interface can have methods and variables, but the methods declared in interface are by default abstract (only method signature and no body).
2. Blueprint of class.
3. If a class implements an interface and does not provide method bodies for all function specified in the interface, then class must be declared abstract.

## Array-Based Stack Implementation

1. Create a stack using an array.
2. Integer variable t – the index of top element in array.
3. Array indices start at 0, so we initialize t to -1.
4. Array-Based Stack implementation is simple and efficient (methods performed in O(1) time)
5. There is an upper bound N, on the size of the stack. The arbitrary value N may be too small for a given application or waste of memory.

## A Growable Array-Based Stack

1. When array gets full, create new array and copy elements
2. **Tight Strategy** (add a constant) f(N) = N + c
3. **Growth Strategy** (double up) f(N) = 2N

## Tight versus Growth Strategy

1. Push Operation = 1 unit
2. Create array and copy elements and add one element

Cost = f(N) + N + 1 units

1. Total cost of phase I =

C\*i = The cost of creating array

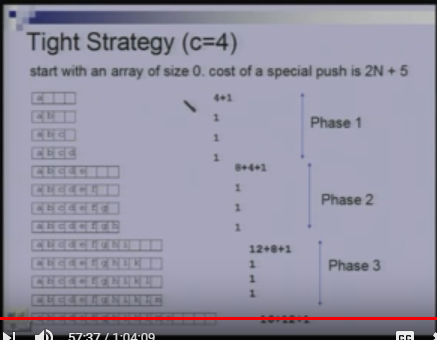
C \* (i – 1) = The cost of copying elements into new array

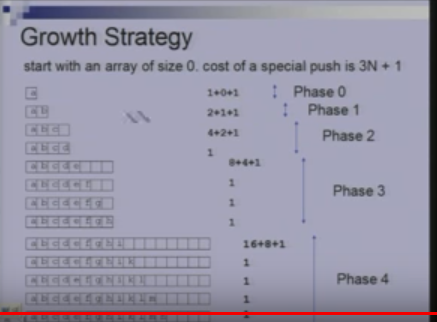
C = The cost of c pushes

Cost of phase I = 2ci

In each phase we do c pushes. Hence for n pushes we need n/c phases.

Total cost of n/c phases = 2c(1 + 2 + 3 …n/c) = O(n^2/c)





1. In phase i, the array has size 2^i
2. Total cost of phase i

2^I = The cost of creating array

2 ^ i-1 = The cost of copying elements into new array

2 ^ I – 1 = The cost of 2^i-1 pushes done in this phase.

Cost of phase I = 2 ^ i+1

1. If we do n pushes then we have logn pushes.
2. Total cost of n pushes = 2 + 4 + 8 + 2 ^ logn + 1 = 4n – 1

The Growth Strategy wins.