DS221 | 19 Sep – 19 Oct, 2017

Data Structures, Algorithms & Data Science Platforms

Yogesh Simmhan

simmhan@cds.iisc.ac.in

What we will cover

- **Data Structures & Algos: 6 lectures**
  - Refresher of data structure basics
  - Some "advanced" topics on trees, graphs, concurrent structures
  - Algorithmic analysis and design patterns
  - Will NOT teach programming
  - 1 programming assignment, **26 Sep** [10 points]
- **Data Science Platforms: 3 lectures**
  - Introduction to Cloud computing, Big Data platforms
  - Apache Spark, tutorial
  - 1 short programming assignment, **10 Oct** [5 points]
- **Mid-term exam, 19 Oct [10 points]**

Class Resources

- Website
  - Schedule, Lectures, Assignments, Additional Reading
  - http://cds.iisc.ac.in/courses/ds221/
- Textbook
  - Data Structures, Algorithms, and Applications in C++, 2nd Edition, Sartaj Sahni*,**
    - http://www.cise.ufl.edu/~sahni/dsaac/
- Other resources
  - The C++ Programming Language, 3rd Edition, Bjarne Stroustrup
  - THE ART OF COMPUTER PROGRAMMING (Volume 1 / Fundamental Algorithms), Donald Knuth
  - Introduction to Algorithms, Cormen, Leiserson, Rivest and Stein
  - www.geeksforgeeks.org/data-structures/

*http://www.tatabookhouse.com/data-structures-algorithms-and-applications-in-c-plus-plus--9788173715228?ver=1519259641
**http://www.flipkart.com/data-structures-algorithms-applications-c-english-2nd/p/itmeyf6jvka3kzdu

Ethics Guidelines

- Students must uphold IISc's Code of Conduct.
  - *Review them!* Failure to follow them **will** lead to sanctions and penalties: reduced or failing grade … **Zero Tolerance!**
- Learning takes place both within and outside the class
  - More outside than inside ☺
- Discussions between students and reference to online material is highly encouraged
- However, you must form your own ideas and **complete problems and assignments by yourself**.
- All works submitted by the student as part of their academic assessment must be their own!

L1: Introduction

Concepts

- **Algorithm**: Outline, the essence of a computational procedure, with step-by-step instructions
- **Program**: An implementations of an algorithm in some programming language
- **Data structure**: Organization of data needed to solve the problem (array, list, hashmap)
- **Algorithmic Analysis:** The expected behaviour of the algorithm you have designed, *before you run it*
- **Empirical Analysis:** The behaviour of the program that implements the algorithm, *by running it*

Why not just run it and see how it behaves?

Limitation of Empirical Analysis

- Need to implement the algorithm
  - Time consuming
- Cannot exhaust all possible inputs
  - Experiments can be done only on a limited to set of inputs
  - May not be indicative of running time for other inputs
- Harder to compare two algorithms
  - Same hardware/environments needs to be used

How do we design an algorithm?

- Intuition
- Mixture of techniques, design patterns
- Experience (body of knowledge)
- Data structures, analysis

How do we implement a program?

- Preferred High Level Language, e.g. C++, Java, Python
- Map algorithm to language, retaining properties
- Use native data structures, libraries

Then why learn about basic data structures?

Algorithm, Data Structure & Language are interconnected

- Algorithms based on specific data structures, their behavior
- Algorithms are limited to the features of the programming language
  - Procedural, Functional, Object oriented, distributed
- Data structures may/may not be natively implemented in language
  - Java Collections, C++ STL, NumPy

Basic Data Structures

Lists

Collections of data

- Data Structures to store collections of primitive data types
  - Primitive types are called `items, elements, instances, values`…depending on context
  - Primitive types can be boolean, byte, integer, etc.
- Properties of the collection
  - Invariants that must be maintained, irrespective of operations
- Operations on the collection
  - Standard operations to create, modify, access elements
- Different **implementations** for same **abstract** collection

Linear List

Type = int, Size = 7

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Item	36	5	75	11	7	19	-1

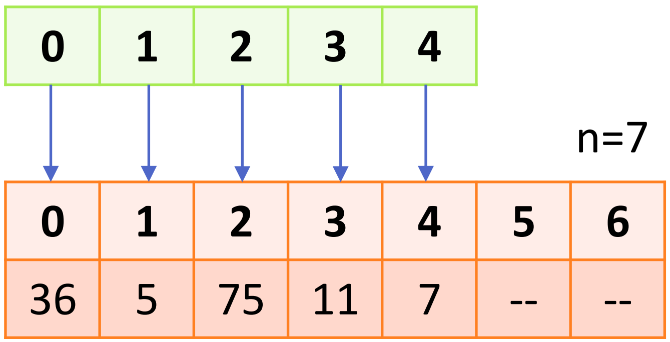
- Properties
  - **Ordered** list of items…precedes, succeeds; first, last
  - **Index** for each item…lookup or address item by index value
  - **Finite size** for the list…can be empty, size may vary
  - Items of **same type** present in the list
- Operations
  - Create, destroy
  - Lookup by index, item value
  - Find size, if empty
  - Add, delete item

1-D Array Representation

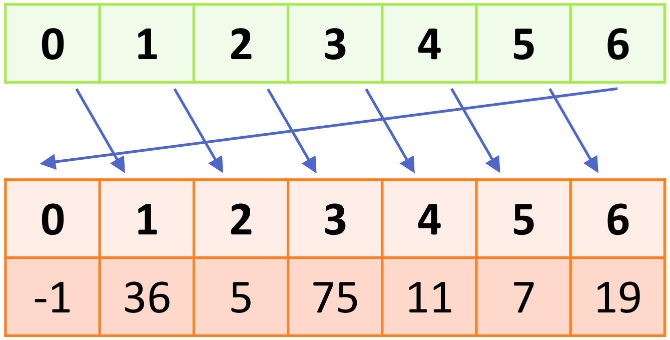
- Implementation of the abstract list data structure using programming language
  - "Backing" Data Structure
- **arrays** are **contiguous** memory locations with **fixed capacity**
- Allow elements of same type to be present at specific **positions** in the array
- **Index** in a **List** can be mapped to a **Position** in the **Array**
  - **Mapping function** from list index to array position

Mapping Function
List index to Array position

- Say **n** is the capacity of the array
- Simple mapping
  - `position(index) = index`



- Wrap-around mapping
  - `position(index) = (position(0)+index) % n`
  - `position(0) = front`



n=7, front=1

List Operations

- `void` **`set`**`(index, item)`
- `item` **`get`**`(index)`
- `void` **`append`**`(item)`
- `void` **`remove`**`(index)`
- `int` **`size`**`()`
- `int` **`capacity`**`()`
- `boolean` **`isEmpty`**`()`
- `int` **`indexOf`**`(item)`

```
class List {        // list with index starting at 1
   int arr[]        // backing array for list
   int capacity     // current capacity of array
   int size         // current occupied size of list

   /**
    * Create an empty list with optional
    * initial capacity provided. Default capacity of 15
    * is used otherwise.
    */
   void create(int _capacity){
     capacity = _capacity > 0 ? _capacity : 15
     arr = new int[capacity]     // create backing array
     size = 0       // initialize size
   }
```

```
    // assuming pos = index-1 mapping fn.
    void set(int index, int item){
      if(index > capacity) { // grow array, double it
        arrNue = int[MAX(index, 2*capacity)]
        // copy all items from old array to new
        // source, target, src start, trgt start, length
        copyAll(arr, arrNue, 0, 0, capacity)
        capacity = MAX(index, 2*capacity) // update var.
        delete(arr) // free up memory
        arr = arrNue
      }
      if(index < 1) {
        cout << “Invalid index:” << index << “Expect >=1”
      } else {
        int pos = index – 1
        arr[pos] = item
        size++
      } // end if
    } // end set()
} // end List
```

List Operations using Arrays

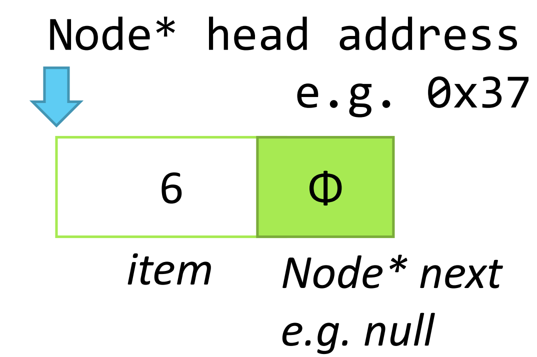
- Increasing capacity
- Start with initial capacity given by user, or default
- When capacity is reached
  - Create array with more capacity, e.g. double it
  - Copy values from old to new array
  - Delete old array space
- Can also be used to shrink space
  - Why?
- **Pros & Cons of List using Arrays**

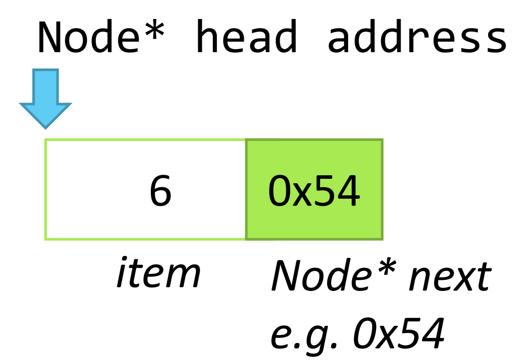
Linked List Representation

- **Problem with array:** Pre-defined capacity, under-usage, cost to move items when full
- **Solution:** Grow backing data structure dynamically when we add or remove ➲ Only use as much memory as required
- *Linked lists* use ***pointers*** to contiguous chain items
  - **Node** structure contains **`item`** and pointer to **`next`** node in List
  - Add or remove nodes when setting or getting items

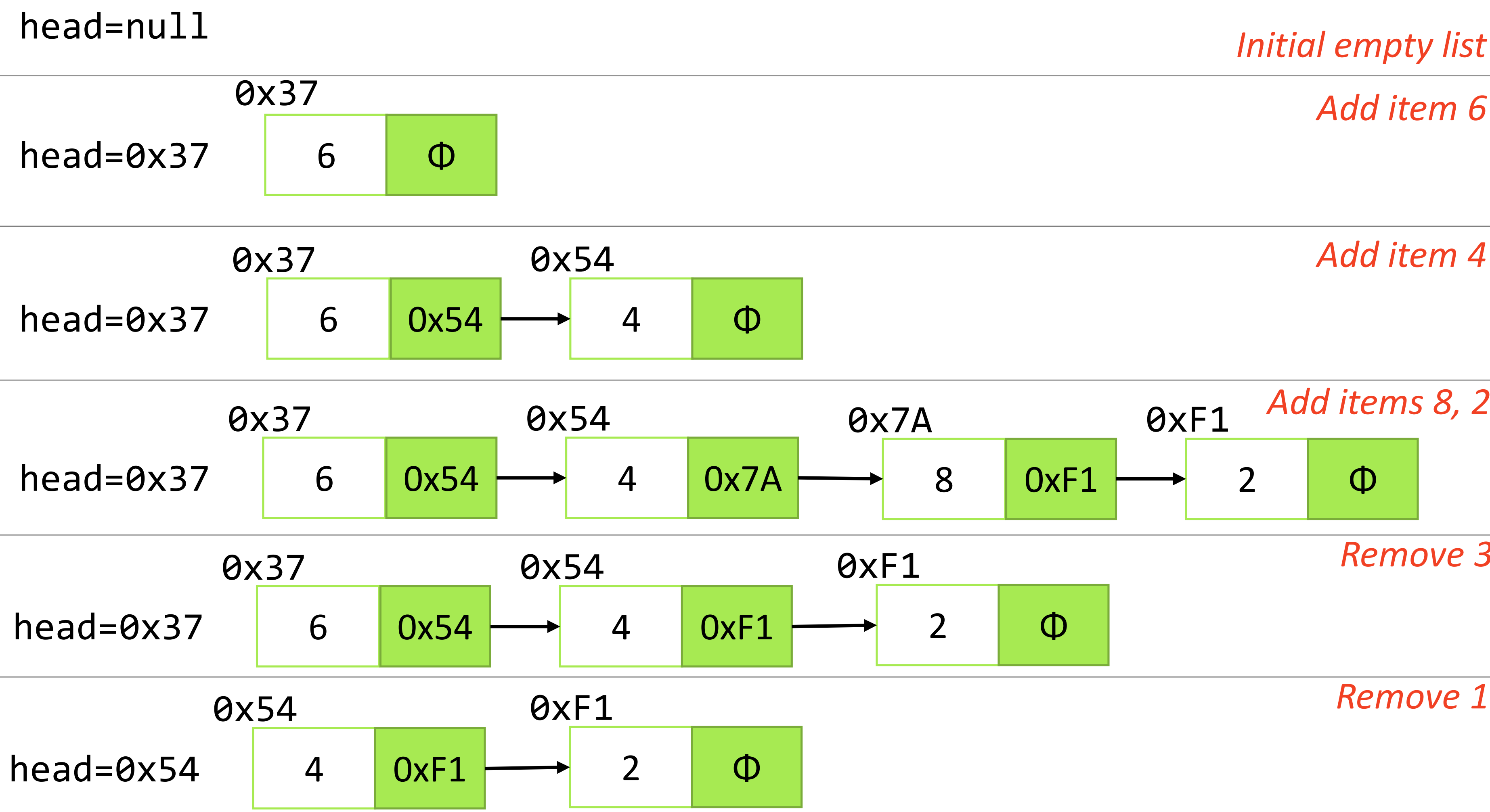
Node & Chain

```
class Node {
   int item
   Node* next
}
class LinkedList {
   Node* head
   int size
   append() {...}
   get() {...}
   set() {...}
   remove {...}
}
```





Linked List Operations



Algorithmic Analysis

Algorithm Analysis

- Algorithms can be evaluated on two performance measures
- Time taken to run an algorithm
- Memory space required to run an algorithm
- …for a given input size
- Later, I/O and Communication complexity
- *Why are these important?*

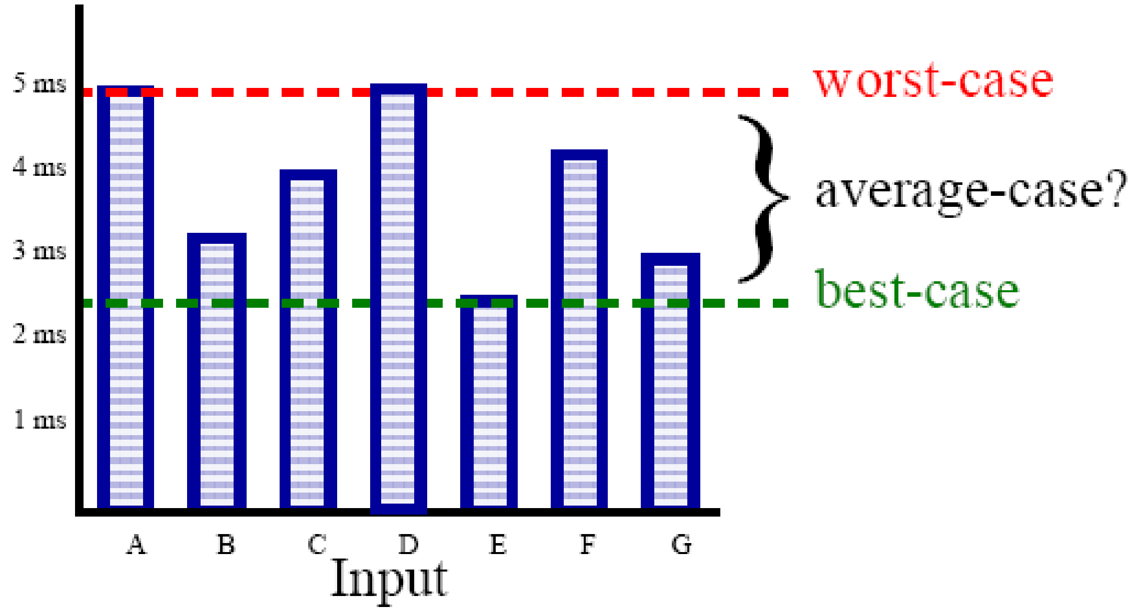
Space Complexity

- *Estimate of the amount of peak memory required for an algorithm to run to completion, <u>for a given input size</u>*
  - Core dumps/OOMEx: Memory required is larger than the memory available on a given system
  - Algorithm design problem OR “memory leaks” in implementation
- Some algorithms may be more efficient if data completely loaded into memory
  - Need to look also at system limitations

Space Complexity

- **Fixed part**: The size required to store certain data/variables, that is independent of the size of the problem:
  - e.g., for all valid words, given a set of letters
  - e.g., etymology for each work in a dictionary
- **Variable part**: Space needed by variables, whose size is dependent on the size of the problem:
  - e.g., number of letters in a scrabble game
  - e.g., text of Shakespeare's plays

Running Time



- Suppose the program includes an *if-then statement that may* execute or not → variable running time
- Typically algorithms are measured by their ***worst case***

General Methodology for Analysis

- Uses High Level Description instead of implementation
- Takes into account for all possible inputs
- Allows one to evaluate the efficiency independent of hardware/software environment

Pseudo-Code

- Mix of natural language and high level programming concepts that describes the main idea behind algorithm
- Control flow
  - If … then …else
  - While-loop
  - for-loop
- Simple data structures
  - Array : A[i]; A[I,j]
- Methods
  - Calls: methodName(args)
  - Returns: return value

```
int arrayMax(int[] A, int n)
  Max=A[0]
  for i=1 to n-1 do
    if Max < A[i]
      then Max = A[i]
  return Max
```

Analysis of Algorithms

- Analyze time taken by Primitive Operations
- Low level operations independent of programming language
  - Data movement (assign..)
  - Control (branch, subroutine call, return…)
  - Arithmetic/logical operations (add, compare..)
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm

Example: Array Transpose

```
function Transpose(A[][], n)
  for i = 0 to n-1 do
    for j = i+1 to n-1 do
      tmp = A[i][j]
      A[i][j] = A[j][i]
      A[j][i] = tmp
    end
  end
end
```

	j=0			j=3
i=0	0,0	0,1	0,2	0,3
	1,0	1,1	1,2	1,3
	2,0	2,1	2,2	2,3
i=3	3,0	3,1	3,2	3,3

Swap

Outer Loop

Estimated time for A[n][n] = (**n(n-1)/2**).(3+2) + 2.n

Inner Loop

Is this constant for a given 'n'?

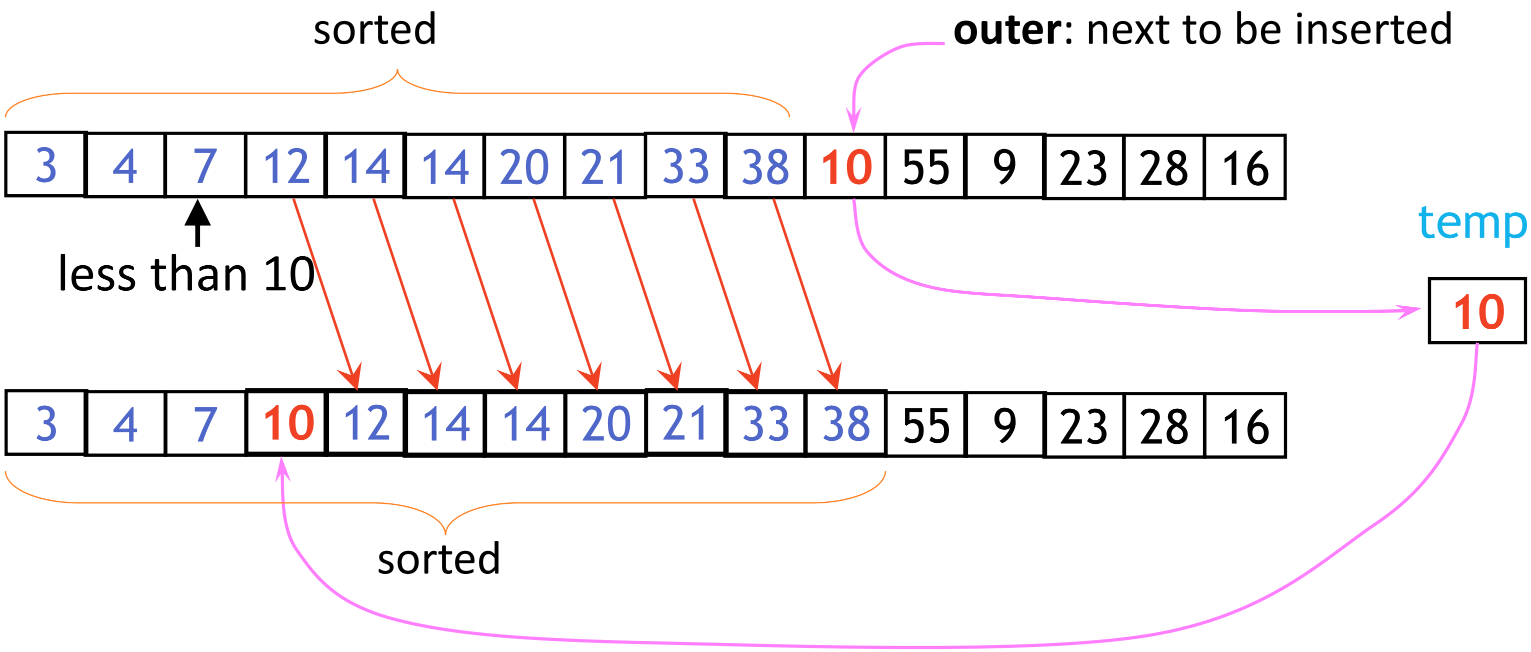
Example: Sorting

- Correctness:
  - For any given input the algorithm stops with the output `{b1 < b2 < b3 … < bn}` which is a permutation of the input `{a1, a2, … an}`
- Running time depends on:
  - Number of elements (n)
  - How partially sorted
  - Algorithm used

Insertion sort

- The outer loop of insertion sort is:
  for (outer = 1; outer < a.length; outer++) {...}
- The invariant is that all the elements to the left of outer are *sorted with respect to one another*
  - For all i < outer, j < outer, if i < j then a[i] <= a[j]
  - This does *not* mean they are all in their final correct place; the remaining array elements may need to be inserted
  - When we increase outer, a[outer-1] becomes to its left; we must keep the invariant true by inserting a[outer-1] into its proper place
  - This means:
    - Finding the element's proper place
    - Making room for the inserted element (by shifting over other elements)
    - Inserting the element

One step of insertion sort



Analysis of insertion sort

- We run once through the outer loop, inserting each of *n* elements; this is a factor of n
- On average, there are $n/2$ elements already sorted
  - The inner loop looks at (and moves) half of these
  - This gives a second factor of $n/4$
- Hence, the time required for an insertion sort of an array of n elements is proportional to $n^2/4$

Analysis of Insertion Sort

# of Sorted Elements	Best case	Worst case
0	0	0
1	1	1
2	1	2
…	…	…
n-1	1	n-1
	n-1	**n(n-1)/2**

Asymptotic Analysis

- **Goal**: to simplify analysis of running time by getting rid of 'details' which may be affected by specific implementation and hardware.
  - Like 'rounding': 1001 = 1000
  - $3n^2=n^2$
- How the running time of an algorithm increases with the size of input in the limit.
  - Asymptotically more efficient algorithms are best *for all but* small inputs.

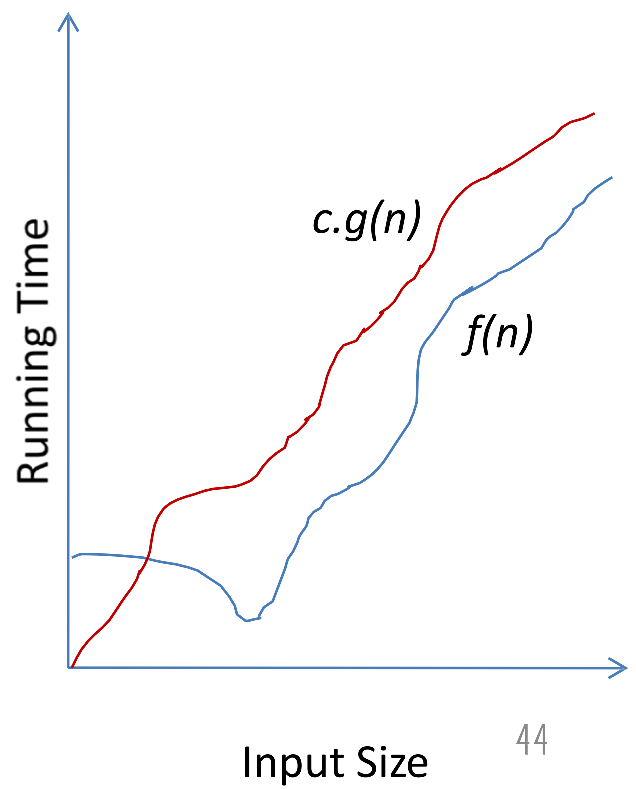
Asymptotic Notation: "Big O"

Definition 3.1 *Let $p(n)$ and $q(n)$ be two nonnegative functions. $p(n)$ is* **asymptotically bigger** *($p(n)$ asymptotically dominates $q(n)$) than the function $q(n)$ iff*

$$\lim_{n \to \infty} \frac{q(n)}{p(n)} = 0 \quad (3.1)$$

TB, Sahni

- ***O*** Notation
  - Asymptotic **upper** bound
  - *f(n)=O(g(n)),* if there exists constants *c* and n_0, s.t.
    - $f(n) \leq c.g(n)$ for $n \geq n_0$
  - *f(n) and g(n) are functions over non negative intergers*
- Used for *worst-case* analysis
  - g(n) is the **asymptotic upper bound** of actual time taken



Asymptotic Notation

- Simple Rule: Drop lower order terms and constant factors
  - `(n(n-1)/2).(3+2) + 2.n is` **`O(n²)`**
  - `23.n.log(n) is` **`O(n.log(n))`**
  - `9n-6 is` **`O(n)`**
  - `6n².log(n) + 3n² + n is` **`O(n².log(n))`**
- Note: It is expected that the approximation should be as small an order as possible

Asymptotic Analysis of Running Time

- Use *O* notation to express number of primitive operations executed as a function of input size.
- Hierarchy of functions

$$1 < \log n < n < n^2 < n^3 < 2^n$$

$\xleftarrow{}$ *Better*

- Warning! Beware of large constants (say 1M).
  - This might be less efficient than one running in time $2n^2$, which is $O(n^2)$

Example of Asymptotic Analysis

- Input: An array X[n] of numbers.
- Output: An array A[n] of numbers s.t A[k]=mean(X[0]+X[1]+...+X[k-1])

```
for i=0 to (n-1) do
   a=0
   for j=0 to i do
      a = a + X[j]
   end
   A[i] = a/(i+1)
end
return A
```

- **Analysis:** running time is $O(n^2)$

A Better Algorithm

```
s=0
for i=0 to n do
  s = s + X[i]
  A[i] = s/(i+1)
end
return A
```

- Analysis: running time is $O(n)$

Asymptotic Notation

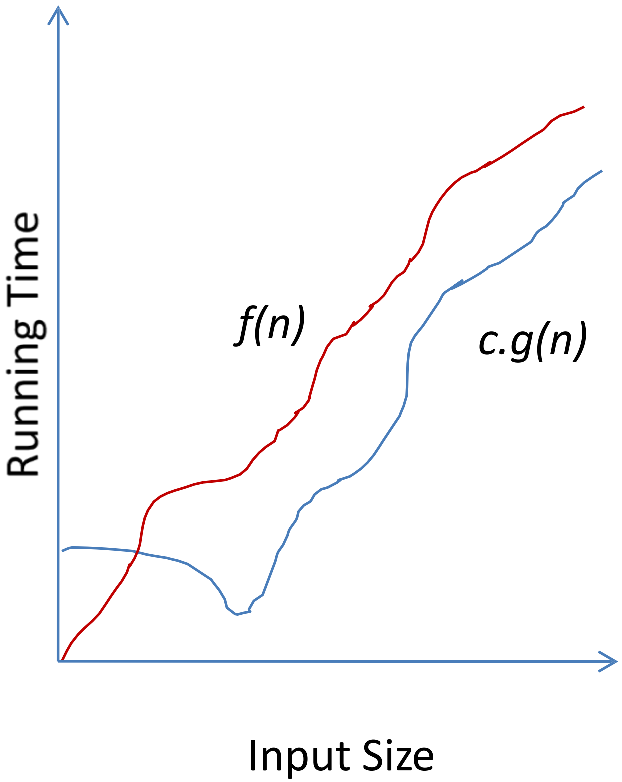
- Special Cases of algorithms
  - Logarithmic $O(\log n)$
  - Linear $O(n)$
  - Quadratic $O(n^2)$
  - Polynomial $O(n^k)$, $k > 1$
  - Exponential $O(a^n)$, $a > 1$

Comparison

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

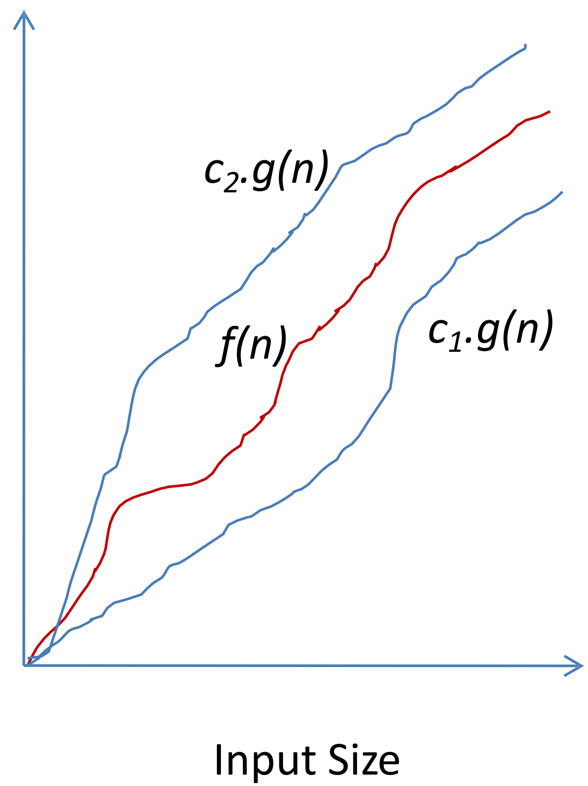
Asymptotic Notation: Lower Bound

- The "big-Omega" Ω notation
  - – asymptotic *lower* bound
  - – $f(n) = \Omega(g(n))$ if there exists const. c and n_0 s.t.
    - **c.g(n) ≤ f(n)** for $n \geq n_0$
- – Used to describe *best-case asymptotic running* times
  - E.g., lower-bound of *searching* an unsorted array; lower bound for *sorting* an array



Asymptotic Notation: Tight Bound

- The "big-Theta" θ-Notation
  - Asymptotically tight bound
  - f(n) = θ(g(n)) if there exists consts. c_1, c_2 and n_0 s.t. $\mathbf{c_1\, g(n) \leq f(n) \leq c_2\, g(n)}$ for $n \geq n_0$
- *f(n) = θ (g(n))* **iff** *f(n)=O(g(n))* and *f(n)=Ω(g(n))*



Small "o"

- ***o*** Notation
  - Asymptotic **strict upper** bound
  - *f(n)=O(g(n)),* if there exists constants *c* and n_0, s.t.
    - *f(n)* < *c.g(n)* for n ≥ n_0

*Similarly small omega, **ω**, is strict lower bound*

Asymptotic Notation

- Analogy with real numbers
  - $f(n) = O(g(n)) \rightarrow f \leq g$
  - $f(n) = \Omega(g(n)) \rightarrow f \geq g$
  - $f(n) = \Theta(g(n)) \rightarrow f = g$
  - $f(n) = o(g(n)) \rightarrow f < g$
  - $f(n) = \omega(g(n)) \rightarrow f > g$

Polynomial and Intractable Algorithms

- **Polynomial Time complexity**
  - An algorithm is said to be polynomial if it is ***O(n^d)***
    - for some integer ***d***
  - Polynomial algorithms are said to be efficient
    - They solve problems in reasonable times!
- ***Intractable Algorithms***
  - **Algorithms for which there is no *known***
    - **polynomial time algorithm.**

Complexity: List using Arrays

- **Storage Complexity**: Amount of storage required by the data structure, relative to items stored
- List using Array: …
- **Computational Complexity**: Number of CPU cycles required to perform each data structure operation
- size(), set(), get(), indexOf()

Complexity: List using Linked List

- Storage Complexity
  - Only store as many items as you need
  - But…
- Computational Complexity
  - set(), get(), remove()
  - indexOf()
- Other Pros & Cons?
  - Memory management, mixed item types

Choosing between List implementations

- When to pick array based List?

- When to pick Linked List?

- Other lists
  - Doubly linked list
  - Sequential lists & Iterators

Tasks

- Self study (Sahni Textbook)
  - Chapter 3 & 4 "Asymptotic Notation" & "Performance Measurement"
  - Chapters 5 & 6 "Linear Lists—Array & Linked Representations"