CHAPTER I

Data Structures and Algorithm Analysis in Java 3rd Edition by Mark Allen Weiss

Aims and objectives of the course

- Aim of the course: understanding of issues involved in program design; good working knowledge of common algorithms and data structures
- Objectives:
  - be able to identify the functionality required of the program in order to solve the task at hand;
  - design data structures and algorithms which express this functionality in an efficient way;
- be able to evaluate a given implementation in terms of its efficiency and correctness.

DATA

- Data as a general concept refers to the fact that some existing information or knowledge is *represented* or *coded* in some form suitable for better usage or processing.

DATA

- Representing information is fundamental to computer science.
- The primary purpose of most computer programs is not to perform calculations, but to store and retrieve information — usually as fast as possible.

DATA STRUCTURE

- In the most general sense, a data structure is any data representation and its associated operations.
- sorted list of integers stored in an array is an example of such a structuring.

ALGORITHM

- In mathematics and computer science, an **algorithm** is a self-contained step-by-step set of operations to be performed

Analysis of algorithms

- Correctness
- Termination
- Time analysis: How many instructions does the algorithm execute?
- Space analysis: How much memory does the algorithm need to execute?

Relation to Algorithms

- Most data structures have associated algorithms to perform operations, such as search, insert, or balance, that maintain the properties of the data structure

- Algorithms and data structures should be thought of as a unit, neither one making sense without the other.

Example

Algorithm *LargestNumber*

Input: A list of numbers L.

Output: The largest number in the list L.

if *L.size = 0* ***return*** *null*

largest ← L[0]

for each *item* ***in*** *L,* ***do***

if *item > largest,*

    then *largest ← item*

return *largest*

Mathematical Review

- Exponents
- Logarithms
- Series
- Modular Arithmetic
- Recursive Definitions
- Function Growth
- Proofs

Exponents

- $X^0 = 1$ by definition
- $X^a X^b = X^{(a+b)}$
- $X^a / X^b = X^{(a-b)}$

Show that: $X^{-n} = 1 / X^n$

- $(X^a)^b = X^{ab}$

Logarithms

- $\log_a X = Y \Leftrightarrow a^Y = X$, $a > 0, X > 0$

  E.G: $\log_2 8 = 3$; $2^3 = 8$
- $\log_a 1 = 0$ because $a^0 = 1$

  $\log X$ means $\log_2 X$

  $\lg X$ means $\log_{10} X$

  $\ln X$ means $\log_e X$,

  where 'e' is the natural number

Logarithms

- $\log_a(AB) = \log_a A + \log_a B$
- $\log_a(A/B) = \log_a A - \log_a B$
- $\log_a(A^n) = n\log_a A$
- $\log_A B = (\log_2 B)/ (\log_2 A)$
- $a^{\log_a x} = x$

Series

- $$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$$

- $$\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$$

These are geometric series

Modular Arithmetic

- A is congruent to B modulo N, $A \equiv B \pmod{N}$, if N divides $A - B$.
- if $A \equiv B \pmod{N}$, then $A + C \equiv B + C \pmod{N}$ and $AD \equiv BD \pmod{N}$
- If N is prime the $ab \equiv 0 \pmod{N}$ is true if and only if $a \equiv 0 \pmod{N}$ or $b \equiv 0 \pmod{N}$.

Recursive Definitions

- **Basic idea:** To define objects, processes and properties in terms of

  simpler objects,

  simpler processes or

  properties of simpler objects/processes.

Recursive Definitions

- **Terminating rule** - defining the object explicitly.
- **Recursive rules** - defining the object in terms of a simpler object.

Examples

- **Factorials** N!

  $$f(n) = n!$$

$$f(0) = 1 \qquad \text{i.e. } 0! = 1$$

$$f(n) = n * f(n-1)$$

$$\text{i.e. } n! = n * (n-1)!$$

Examples

- Fibonacci numbers

$$F(0) = 1$$

$$F(1) = 1$$

$$F(k+1) = F(k) + F(k-1)$$

1, 1, 2, 3, 5, 8, ….

RULES OF RECURSION

- **Base Case** : You must always have some base case which can be solved without recursion
- **Making Progress**: For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress towards the base case
- **Design rule**: Assume that all recursive calls work.

Principles and Rules

- Base cases: you must always have some base cases, which can be solved without recursion.
  – Solving the recursive function $f(x) = 2f(x-1) + x^2$
  – Example 1

```
int f (int x) {
  if (x == 0)
    return 0
  else
    return 2*f(x-1)+x*x
```

Base Case: since at this point recursion is not needed to determine a return value.

- Note a base case generally kills the recursion by not allowing future recursive calls.

Example 2

```
PrintList (node ptr){
    if (ptr != null) {
        print(ptr.data);
        PrintList(ptr.next);
    }
}
```

If statement prevents recursive part from being called.

Principles and Rules Cont...

- Making progress: For the cases that are solved recursively, the recursive call must always be to a case that makes progress toward a base case

```
int f (int x) {
  if (x == 0)
    return 0
  else
    return 2*f(x-1)+x*x
```

X is reduced toward the base case.

```
PrintList (node ptr){
   if (ptr != null) {
       print(ptr.data);
       PrintList(ptr.next);
   }
}
```

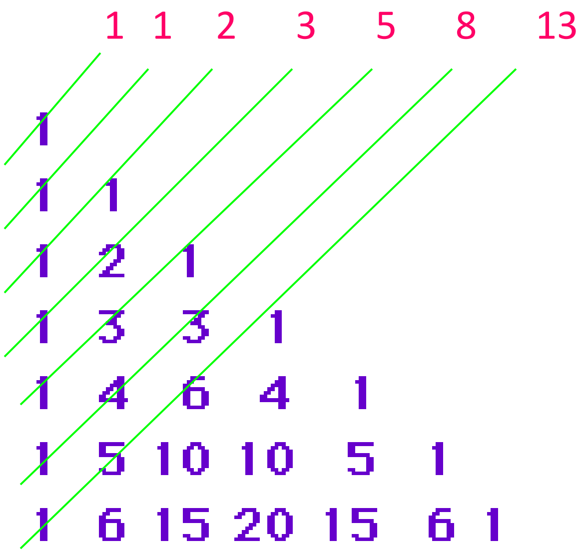
We assume that ptr.next will take us toward the end of the list, hence a null pointer.

Principles and Rules Cont.

- Design rule: Assume that all recursive calls work.
  - In general the machine uses its own stack to organize the calls.
  - Very difficult to trace the calls.
  - Important to ensure the code adheres to the first two principles, this rule then takes care of itself.

Principles and Rules Cont.

- Compound interest rule: Never duplicate work by solving the same instance of a problem in separate recursive calls.




```
fib( int n) {
  if (n <= 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

Principles and Rules Cont.

- n = 3 the first instance of fib calculates:
  - fib(n-1) ==> fib(2) and fib(n-2) ==> fib(1) line 1
- fib(2) results in instances
  - fib(1) and fib(0) line 2
- Notice:
  - Instance of fib(1) is now known form line 2
  - But fib(1) in line 1 has yet to be processed
  - So fib(1) is recalculated.
- Results in:
  - Slow algorithms
  - Increases the magnitude unnecessarily

What the Machine does - Call Entry

- Each program has a "Call Stack".
  - When a method is called the state of the system is stored in a Call Frame.
  - This includes all local variables and state information.
    - It is like taking a snap shot of the system.

```
Method B{…}
Method A {
  Call Method B
}
Main(){
   Call Method A
}
```

Call Stack with Call Frames for each suspend process

System state and local variables of Main Method	Current state of Method A is stored when Method B is Called, including local variables	

Method A is Called

Method B is called, forcing Method A to be suspended

What the Machine does - Call Return

- When a method returns:
  - The call frame is removed from the stack
  - Local variables are restored
  - System continues from this returned state.
- Since a call frame stores a system state, we can say a history is also stored.
  - Previous values, Previous locations.
  - History is a state of suspended animation which can be woke at the correct time

Proofs

- Direct proof
- Proof by induction
- Proof by counterexample
- Proof by contradiction
- Proof by contraposition

Direct Proof

Based on the definition of the object / property

Example:

Prove that if a number is divisible by 6 then it is divisible by 2

Proof: Let m divisible by 6.

Therefore, there exists q such that m = 6q

6 = 2 . 3

m = 6q = 2.3.q = 2r, where r = 3q

Therefore m is divisible by 2

Proof by Induction

We use proof by induction when our claim concerns a sequence of cases, which can be numbered

Inductive base:

Show that the claim is true for the smallest case, usually $k = 0$ or $k = 1$.

Inductive hypothesis:

Assume that the claim is true for some k

Prove that the claim is true for $k+1$

Example of Proof by Induction

Prove by induction that

$S(N) = \sum_{i=0 \text{ to } N} 2^i = 2^{(N+1)} - 1$, for any integer $N \geq 0$

1. **Inductive base**

   Let $n = 0$. $S(0) = 2^0 = 1$

   On the other hand, by the formula $S(0) = 2^{(0+1)} - 1 = 1$.

   Therefore the formula is true for $n = 0$

2. Inductive hypothesis

   Assume that $S(k) = 2^{(k+1)} - 1$

   We have to show that $S(k+1) = 2^{(k+2)} - 1$

By the definition of $S(n)$:

$S(k+1) = S(k) + 2^{(k+1)} = 2^{(k+1)} - 1 + 2^{(k+1)} = 2.\ 2^{(k+1)} - 1 = 2^{(k+2)} - 1$

Example 2

$$if\ N \geq 1\ then\ \sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}$$

$$\sum_{i=1}^{N+1} i^2 = \sum_{i=1}^{N} i^2 + (N+1)^2$$

$$\sum_{i=1}^{N+1} i^2 = \frac{N(N+1)(2N+1)}{6} + (N+1)^2$$

$$= (N+1)\left[\frac{N(2N+1)}{6} + (N+1)\right]$$

$$= (N+1)\,\frac{2N^2 + \ 7N + 6}{6}$$

$$= \frac{(N+1)(2N^2 + 4N + 3N + \ 6)}{6}$$

$$= \frac{(N+1)[(N+1)+1][2(N+1)+1]}{6}$$

$$= \frac{(N+1)(N+2)(2N+3)}{6}$$

Proof by Counterexample

Used when we want to prove that a statement is false.
Types of statements: a claim that refers to all members of a class.

EXAMPLE: The statement "all odd numbers are prime" is false.

A counterexample is the number 9: it is odd and it is not prime.

Proof by Contradiction

Assume that the statement is false, i.e. its negation is true.

Show that the assumption implies that some known property is false - this would be the contradiction

Example: Prove that there is no largest prime number

Proof by Contraposition

Used when we have to prove a statement of the form $P \to Q$.

Instead of proving $P \to Q$, we prove its equivalent $\sim Q \to \sim P$

Example: Prove that if the square of an integer is odd then the integer is odd

We can prove using direct proof the statement:

If an integer is even then its square is even.