Lecture 8

How do we analyze algorithms? Model of Computation (RAM Model), Mathematical Analysis of Non-Recursive Algorithms, and Worst, Best & Average Case Behavior of Algorithms.

How do we analyze algorithms?

- *We need to define a number of objective measures.*

(1) Compare execution times?

***Not good**: times are specific to a particular computer*

(2) Count the number of statements executed?

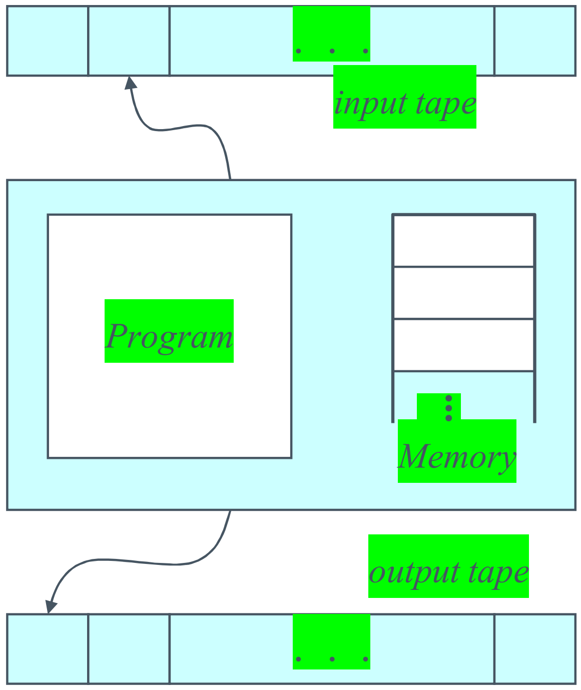
***Not good**: number of statements vary with the programming language as well as the style of the individual programmer. (see example on next slide)*

The RAM Model

- *Random Access Machine (not R.A. Memory)*
- *An idealized notion of how the computer works*
  - *Each "simple" operation (+, -, =, if) takes exactly 1 step.*
  - *Each memory access takes exactly 1 step*
  - *Loops and method calls are not simple operations but depend upon the size of the data and the contents of the method.*
- *Measure the run time of an algorithm by counting the number of steps.*

Random Access Machine

- *A Random-Access Machine* (RAM) consists of:
  - a fixed *program*
  - an unbounded *memory*
  - a read-only *input tape*
  - a write-only *output tape*
- Each *memory register* can hold an arbitrary integer (*).
- Each *tape cell* can hold a single symbol from a finite *alphabet* s.



Instruction set:

$x \leftarrow y$, $x \leftarrow y$ {+, -, *, div, mod} z

goto *label*

if y {<, ≤, =, ≥ ,> , ≠} z goto *label*

$x \leftarrow$ input, output $\leftarrow y$

halt

Space Complexity

- *The amount of memory required by an algorithm to run to completion*
  - *The term memory leaks refer to the amount of memory required is larger than the memory available on a given system*
- *Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.*
- *Auxiliary Space is confused with Space Complexity. But Auxiliary Space is the extra space, or the temporary space used by the algorithm during its execution.*
- Space Complexity = Auxiliary Space + Input space

Space Complexity (Cont !!!)

- ***Fixed part:*** *The size required to store certain data/variables, that is independent of the size of the problem:*
  - *Such as int a (2 bytes, float b (4 bytes) etc*

- ***Variable part:*** *Space needed by variables, whose size is dependent on the size of the problem:*
  - *Dynamic array a[]*

Space Complexity (cont'd)

- *S(P) = c + S(instance characteristics)*
  - *c = constant*
- ***Example:***

```
void float sum  (float* a, int n)
{
    float s = 0;
     for(int i = 0; i<n; i++) {
        s+ = a[i];
     }
     return s;
}
```

Constant Space:

one for n, one for a [passed by reference!], one for s, one for I , constant space=c=4

Running Time of Algorithms

- *Running time*
  - *depends on input size n*
    - *size of an array*
    - *polynomial degree*
    - *# of elements in a matrix*
    - *# of bits in the binary representation of the input*
    - *vertices and edges in a graph*
  - *number of primitive operations performed*
- *Primitive operation*
  - *unit of operation that can be identified in the pseudo-code*

Steps To determine Time Complexity

Step-1. Determine how you will measure input size. Ex:

- *N items in a list*
- *N x M table (with N rows and M columns)*
- *Two numbers of length N*

Step-2. Choose the type of operation (or perhaps two operations)

- *Comparisons*
- *Swaps*
- *Copies*
- *Additions*

Note: *Normally we don't count operations in input/output**.***

Steps To determine Time Complexity (Cont !!!)

Step-3. Decide whether you wish to count operations in the

- ***Best case? -*** *the fewest possible operations*
- ***Worst case? -*** *the most possible operations*
- ***Average case?*** *This is harder as it is not always clear what is meant by an "average case". Normally calculating this case requires some higher mathematics such as probability theory.*

Step-4. For the algorithm and the chosen case (best, worst, average), express the count as a function of the input size of the problem.

Mathematical Analysis of Non-Recursive Algorithms, and Worst, Best & Average Case Behavior of Algorithms

Primitive Operations in an algorithm

- *Assign a value to a variable (i.e. a=5)*
- *Call a method (i.e. method())*
- *Arithmetic operation (i.e. a*b, a-b*c)*
- *Comparing two numbers (i.e. a<=b, a>b &&a>c)*
- *Indexing into an array (i.e. a[0]=5)*
- *Following an object reference (i.e. Test obj)*
- *Returning from a method (i.e. return I)*

Types of Algorithm Complexity

- *Worst Case Complexity:*
  - *the function defined by the maximum number of steps taken on any instance of size n*
- *Best Case Complexity:*
  - *the function defined by the minimum number of steps taken on any instance of size n*
- *Average Case Complexity:*
  - *the function defined by the average number of steps taken on any instance of size n*

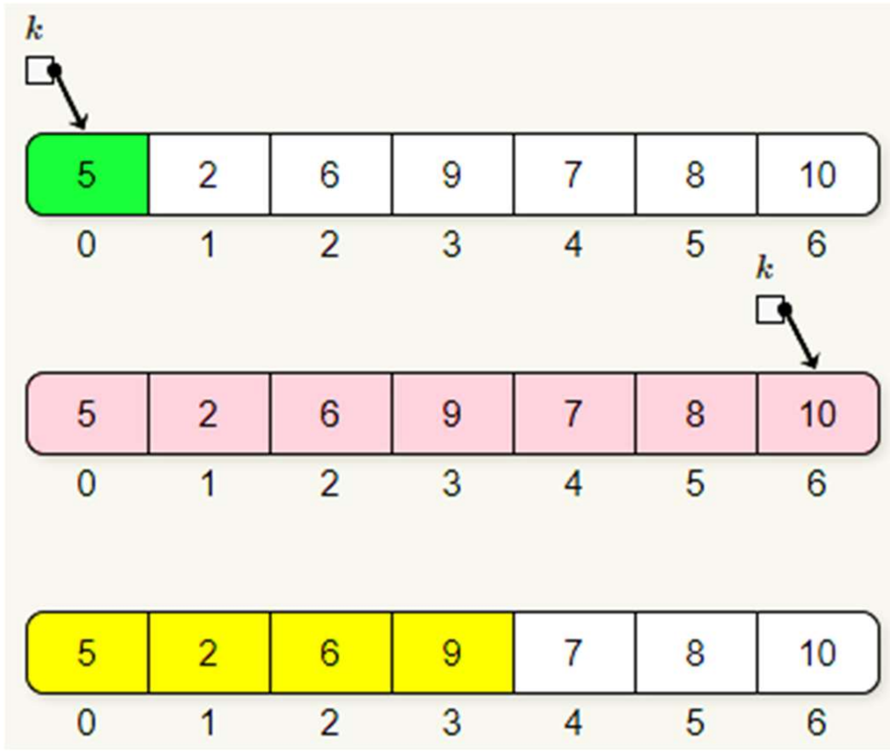
Types of Algorithm Complexity
(Example: Linear Search)

5	2	6	9	7	8	10

- *Worst Case Complexity:*
  - *You want to search 1 in above array which is at location N*
  - *You need N steps*
- *Best Case Complexity:*
  - *You want to search 5 in above array which is at location 1*
  - *You need 1 steps*
- *Average Case Complexity:*
  - *You want to search 2 or 9 etc in above array*
  - *You need 3 steps for 2 and 5 steps for 9*

π

Visual representation of Best, Worst and Average case

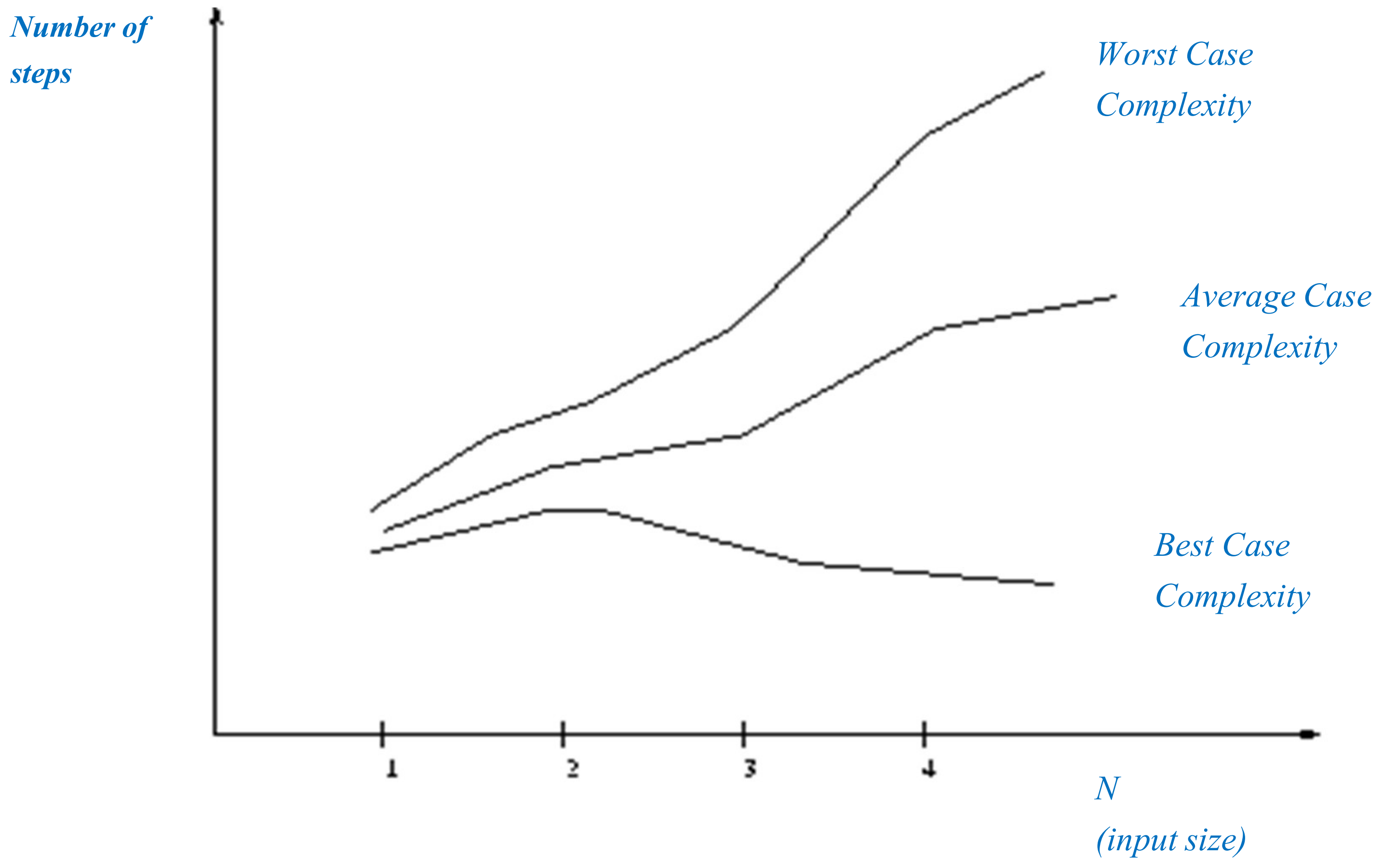


Best Case. A single comparison is performed.

Worst Case. n comparisons are performed.

Average Case. $\frac{n+1}{2}$ comparisons are performed.

Best, Worst, and Average Case Complexity



Relationship between complexity types and running time of Algorithms

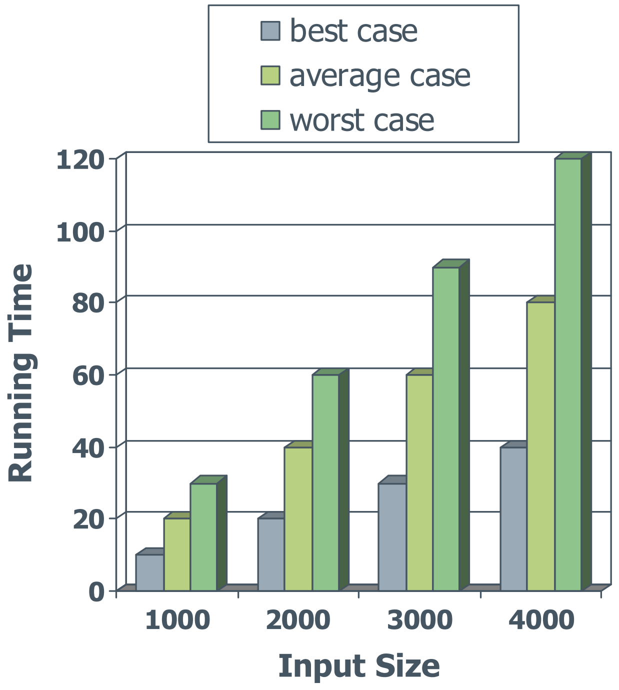
- ***Worst case***
  - *Provides an upper bound on running time*
  - *An absolute guarantee that the algorithm would not run longer, no matter what the inputs are*
- ***Best case***
  - *Provides a lower bound on running time*
  - *Input is the one for which the algorithm runs the fastest*

$$\boxed{Lower\ Bound \le Running\ Time \le Upper\ Bound}$$

- ***Average case***
  - *Provides a prediction about the running time*
  - *Assumes that the input is random*

Running Time

- *Most algorithms transform input objects into output objects.*
- *The running time of an algorithm typically grows with the input size.*
- *Average case time is often difficult to determine.*
- *We focus on the worst-case running time.*
  - *Easier to analyze*
  - *Crucial to applications such as games, finance and robotics*



Homework

- *Exercise-1.*
  - *Write a pseudo code which find the sum of two 3*3 matrices and then calculate its running time.*
- *Exercise-2.*
  - *Write a pseudo code which read a number N and print whether it is prime or not . After that, calculate the run time complexity*

Non-Recursive Algorithm: Few Examples

```
Algorithm X (n)
        i ← 5
        j ← …

        while ( i < …)
            …
        k ← Algorithm Y(…)
          …
        for (…)
           while (…)
              …
      return …
```

Complexity of Algorithm X =

MaxElement

```
MaxElement(A[1..n])
  maxval ← A[1]
  for i ← 2 to n do
      if A[i] > maxval
            maxval ← A[i]
  return maxval
```

```
UniqueElement(A[1..n])

 for i ← 1 to n-1 do
            for j ← i+1 to n do{
            if A[i] = A[j]
                       return false}
   return true
```

MatrixMultiplication

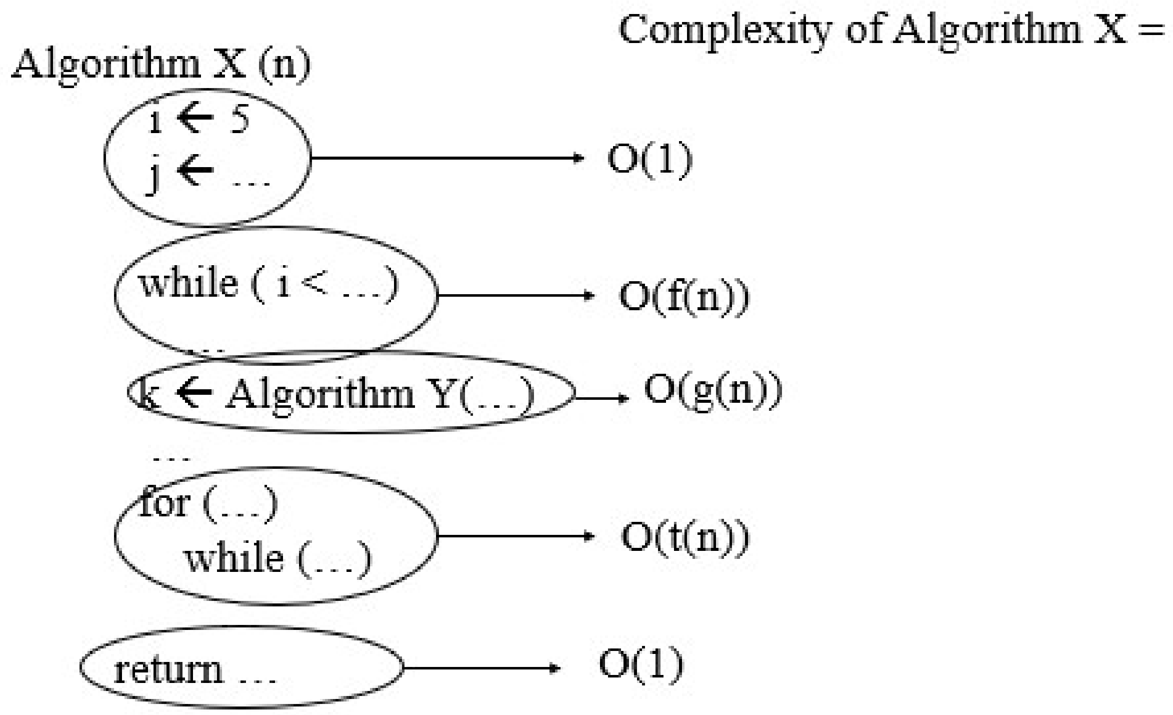
```
UniqueElement(A[1..n,1..n], B[1..n,1..n])

for i ← 1 to n do
      for j ← 1 to n do
          C[i,j] ←0
          for k ← 1 to n do
               C[i,j] ← C[i,j] + A[i,k] *B[k,j]

  return C
```

Non- Recursive Algorithm

Non-Recursive Algorithms



Recursive Algorithms

```
Algorithm X (n, …)
    if (n = 0) return value        ⟶  termination
        while ( i < …)             ⟶  f(n)
            …
        return Algorithm X(n_1)        // where n_1 < n
```

Complexity of Algorithm X = T(n)

$$T(n) = f(n) + T(n_1)$$
$$T(0) = 1$$

//n is the size of the input

Solving a Recursive Equation

1. Make a few evaluations of T(n) for a few values n_1, n_2, n_3 according to the recursive call
2. Deduce a pattern for T(n)
3. Compute the number of times, k, the recursive call is made until the termination condition (e.g.,. T(0))
4. Use 1 and 3 for obtaining a final equation T(n)
5. Solve the equation T(n)

SumElement

```
SumElement(A[1..n], i)
    If  (i = n) then  return A[n]
        Else
            return  A[i]  + sumElement(A,i+1)
```

Thank You!!!

Have a good day