# Advanced Analysis of Algorithm

Department of Computer Science
Swat College of Science & Technology

CS Course : Advanced Analysis of Algorithm
Course Instructor : Muzammil Khan

# Chapter 3

Complexity Analysis

## Complexity Analysis

- Traditionally
  - The running time of a program described as a *function of the size of the input*
  - For example,
    - In a sorting problem, the input size is the number of items *n* in array
  - Input size can also contain more than one parameter e.g. for a graph
- The running time of an algorithm on a particular input is *the number of primitive operations or steps executed*
  - The notion of step should be taken in a machine independent way



## Example

- As an example,
  - Take the pseudo code for an algorithm that finds the maximum element in an array of size *n*
- Problem
  - Develop an algorithm to find maximum element in an array of size *n*
  - Analyze the algorithm for
    - Time efficiency
    - Space efficiency
    - Correctness



## Algorithm in Plain Text

- Steps
- Step #1
  - First element of the array Store in variable ***max***
- Step #2
  - Scan array by comparing ***max*** with other elements
- Step #3
  - Replace ***max*** with a larger element
    - If the ***max*** is not last value, then repeat step 2 & 3
- Step #4
  - Return value held by ***max***



## Pseudocode

- The function **FIND-MAX** finds the maximum element in an array.
  - The array **A**, of size ***n***, is passed as argument to the function

```
FIND-MAX ( A[1..n] )
1  max ←A[1]                    ► Store first array element into variable max
2  for j←2 to n do              ► Scan remaining elements
3      if ( A[j] > max )        ► Compare an element with max
4          then max ← A[j]      ► Replace max with a larger element
5  return max                   ► Return maximum element
```

## Primitive Operations

| # | Statement | Unit costs | Remarks |
|---|---|---|---|
| 1 | *max ←A[1]* | Ca<br>Cs | Cost of accessing *A[1]*<br>Cost of storing *A[1]* into *max* |
| 2 | ***for*** *j←2 to n* ***do*** | Cs<br>Cc<br>Ci | Cost of storing 2 into *j*<br>Cost of comparing index *j* with *n*, and branching<br>Cost of incrementing *j* |
| 3 | ***if*** ( A[j] > max ) | Ca<br>Cc | Cost of accessing *A[j]*<br>Cost of comparing *A[j]* with *max*, and branching |
| 4 | ***then*** *max ← A[j]* | Ca<br>Cs | Cost of accessing *A[j]*<br>Cost of storing storing *A[j]* into max |
| 5 | ***return*** *max* | Cr | Cost of returning maximum element |



## Primitive Operations (Cont...)

| # | Statement | Unit Cost | Operations Count | Total Cost |
|---|---|---|---|---|
| 1 | *max ←A[1]* | Ca<br>Cs | 1<br>1 | Ca + Cs |
| 2 | ***for*** *j←2 to n* ***do*** | Cs<br>Cc<br>Ci | 1<br>n-1<br>n-1 | Cs + (n-1).Cc +(n-1).Ci |
| 3 | ***if*** *( A[j] > max )* | Ca<br>Cc | n-1<br>n-1 | (n-1).Ca + (n-1).Cc |
| 4 | ***then*** *max ← A[j]* | Ca<br>Cs | k<br>k | (Ca + Cs). k<br>$0 \le k \le n$ ( k depends on input array data) |
| 5 | ***return*** *max* | Cr | 1 | Cr |

$T(n) = A + B.k + C.n, \quad where\ 0 \le k \le n$



## Time Complexity

- Time complexity
  - Three type of time complexity
    - Best running time (best case running time)
    - Average running time
    - Worst running time
- Running time of the algorithm for
  - Finding maximum value in the array of length *n* is

    $T(n) = A + B.k + C.n, \quad where\ 0 \le k \le n$

    - Hence *k* is the number of times the statement ***max ← A[j]*** executed



## Best Case Running Time

$T(n) = A + B.k + C.n, \quad where\ 0 \le k \le n$

- Hence *k* is the number of times the statement ***max ← A[j]*** executed

- Best Case
  - Best case occurs when the statement is not executed at all
    - This happen when maximum value is at ***first*** position
    - In this case $k = 0$
  - Best (minimum) running time will be

    $$T_{best}(n) = A + C.n$$



## Average Case Running Time

$T(n) = A + B.k + C.n, \quad where\ 0 \le k \le n$

- Hence *k* is the number of times the statement ***max ← A[j]*** executed

- Average Case
  - Average case occurs when the statement is executed on average $n / 2$ times
    - This happen when maximum value lies in the ***middle*** of the array
    - In this case $k = n / 2$

    $$T_{average}(n) = A + (B / 2 + C).n$$
  - The better average case time can be analysis by *probabilistic analysis*



## Worst Case Running Time

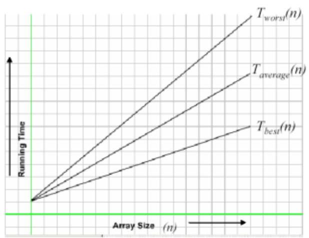
$T(n) = A + B.k + C.n, \quad where\ 0 \le k \le n$

- Hence *k* is the number of times the statement ***max ← A[j]*** executed

- Worst Case
  - In this case the statement is executed $n$ times
    - This happen when maximum value lies at the ***last*** position of the array
    - In this case $k = n$
  - The worst case (maximum) time will be

    $$T_{worst}(n) = A + (B + C).n$$

## Comparison of Running Times






## Space Efficiency

- Space analysis of algorithm that find maximum element in the array is simple and straight
  - Determining the space as a function of **Array Size**
  - Space required by the program
- The amount of storage requirement of the array
  - Depends on nature of the data
    - Integers, Floating point, String etc…
- Space increases in direct proportion to array size
- So $S(n) = \underbrace{A}_{\text{Program space requirement}} + \underbrace{B.n}_{\text{Array space requirement}}$



## Correctness - Loop Invariant Technique

- Several standard algorithms are based on
  - *One or More iterative computations using loop structures*
  - The correctness of such algorithm is established by proving the correctness of loops
  - One such technique is called **Loop Invariant** Techniques
- Loop Invariant
  - Is set of conditions or relationship that is either true or false during loop execution
  - Depends on nature of the problem being analyze



## Correctness (Cont...)

- Loop Invariant Method
  - The **algorithm is correct** if we establish the following
  - **Initialization**
    - Loop invariant is true prior to execution of first iteration
  - **Maintenance**
    - Loop invariant is true prior to some iteration, will remain true before next iteration
  - **Termination**
    - After termination of the loop, the post condition can be evaluated as true



## Correctness (Cont...)

- The correctness of FIND-MAX algorithm, will establish by loop invariant techniques

```
FIND-MAX(A)
  max ← A[1]
  for j←2 to n do
    if( A[j] > max)
      then   max ← A[j]
  return max
```

- Define loop invariant S
  - S = After *kth* iteration, the variable MAX holds the largest value of the first *k* element of the array
- Now
  - Consider the loop invariant method



## Correctness (Cont...)

```
FIND-MAX(A)
  max ← A[1]
  for j←2 to n do
    if( A[j] > max)
      then   max ← A[j]
  return max
```

- Initialization
  - *Condition requires that the prior the first iteration the statement S should be true*
  - So; this is trivially (vacuously) true
  - Because
    - The max contain 1st element of the array &
    - Loop start with index 2
- Maintenance
  - *Condition requires that if S is true before an iteration of loop, it should remain true before the next iteration*
  - Can be verified as if max holds the largest value of k element, then it holds the largest of k+1 element

## Correctness (Cont...)

```
FIND-MAX(A)
  max ← A[1]
  for j←2 to n do
    if( A[j] > max)
      then   max ← A[j]
  return max
```

- Termination
  - *Condition requires that the post condition should be true*
  - *max* should return the maximum value of the array
  - The loop terminates when *j* exceeds *n*



## Complexity Analysis

- Problem
  - Adding array element
- Algorithm

| ARRAY-ADD (A, n) | Cost | Times |
|---|---|---|
| result ← 0 | $c_1$ | 1 |
| for ( i ←1; i <= n; i++) | $c_2$ | n+1 |
| result ← result + A[i] | $c_3$ | n |

- To compute T(n) the running time of the algorithm,
  - Sum up, product of cost and time column
  - $T(n) = c_1+c_2(n+1)+c_3(n) = A + Bn$
    - Where a and b are constants that depend on $c_i$ )



## Complexity Analysis (Cont...)

- In the previous example
  - There are always *n* passes through the for loop
  - No matter what the value of the numbers/elements in the array
- This is known as the **Every-case running time**
  - Means
    - Best = Average = Worst case running time
- Other examples include ….
  - Consider the example of search in an array



## Complexity Analysis (Cont...)

- Problem
  - Search Array Members
- Algorithm

| ARRAY-SEARCH (A, n, key) | cost | Times |
|---|---|---|
| For ( i ←1; i <= n; i++) | $c_1$ | ? |
| if A[i]=key | $c_2$ | ? |
| return i | $c_3$ | ? |
| return i | $c_4$ | ? |

- Computing Running Time
  - Worst Case Analysis
    - The loop will execute maximum number of time i.e. $n + 1$
    - So $T(n) = A + Bn$ or $W(n) = A + Bn$ or $T(n)_w = A + Bn$



## Complexity Analysis (Cont...)

- Average Case Analysis
  - It is more difficult to analyze the average case than the worst case
  - To compute average case complexity
    - We need to assign probabilities
  - In case of linear search
    - Equal probabilities are assigned to all array slots i.e. the key is equally likely in any array slot
      - Assuming that key has equal probability ***1/n*** of being in any position &
      - Unit cost is ***c***
  - So $1c.1/n + 2c.1/n + 3c.1/n + \ldots + nc.1/n$
    $= c/n\ (1+2+\ldots+n)$
    $= cn\ (n+1)/2n$
    $= c(n+1)/2$



## Complexity Analysis (Cont...)

- So, average case analysis
  $T(n)=c(n+1)/2$ or $A(n)=c(n+1)/2$
- Best Case Analysis
  - When value is found at first location
  - So
    $T(n) = 1$ or $B(n) = 1$
- *Worst-case and average-case analysis are done much more often than best-case analysis*

## End

- End of chapter

- You may have quiz next weak