Data Structucutres and Algorithms SWE2001

Dr. Ramesh Ragala

February 24, 2022

Performance Analysis

- Precise way of Analysing Algorithms is needed to classify some good algorithms.
- Mainly two factors for judging algorithms that have a more direct relationship to performance.
  - Running time of algorithm and data structure operations.
  - Space utilization for each operation of an algorithm.
- Running time is a good measurement.
- The performance of a program means the amount of computer memory and time needed to run a program.
- There are two approaches to determine performance of a program.
  - Analytical Method
  - Experimental Method

Performance Analysis

- Performance Evaluation can be done in two phases
  - Priori Estimate or Apriori Analysis or Perform Analysis
  - Posteriori Testing or Empirical Method or Performance Measurement
- Priori Estimate:
  - Estimating time and space utilization of an algorithm during execution time.
  - @ algorithmic level.
  - Uses Analytical approach to calculate time and space requirement of the algorithm.
  - This Analytical Model uses RAM (Random Access Machine).

Random Access Model

- **Primitive Operations:** Set of High Level operations, which are independent from programming language and available in Pseudo-Code.
- These Primitive operations corresponds to low-level instructions with an execution time that depends on hardware and software environment only.
- Some of the Primitive Operations are:
  - Assigning a value to a variable
  - Calling a method
  - Performing a arithmetic Operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a method
- Counting primitive operation $\rightarrow$ computational model $\rightarrow$ Random Access Model

- **Performance Measurement:**
  - Measuring or calculating the Time and Space requirement of the algorithm while executing on ideal machine.
  - It gives accurate results.
  - The results produced by this approach varies based on the hardware and software environment of Ideal machine.
  - So it is very difficult to accepted the those results.

- **Time Complexity:**
  - The Amount of Computer Time it needs to run to Completion.
- **Space Complexity:**
  - The Amount of Computer Memory it needs to run to Completion.

Time Complexity

- The time **T(P)** taken by Program-P = **Compile Time** + **Run Time**.
- Compiled Program can be run many times without re-compilation.
- Compile time does not depend on the instance Characteristics. $\rightarrow$ **neglect**
- Run time can be denoted as $\mathbf{t}_P$ (Instance Characteristics)
- Factors depends on $\mathbf{t}_P$ are not know in advance $\rightarrow$ **estimate**
- Once we knew the compiler characteristics $\rightarrow$ addition, multiplications etc those would used to made Program-P.
- So the Expression is :
- $\mathbf{t}_P = \mathbf{c}_a\mathbf{ADD(n)} + \mathbf{c}_s\mathbf{SUB(n)} + \mathbf{c}_m\mathbf{MUL(n)} + ...$
  - n $\rightarrow$ Instance Characteristics; c_a,c_s,c_m are denotes the time needed for Addition, Subtraction, Multiplication ...
  - ADD, SUB, MUL are functions, whose values are number of additions, subtraction, multiplications etc

Time Complexity

- Obtaining and following such formula for estimating Time Complexity is Difficult
- **Another Approach:** **Step Count** $\rightarrow$ counts only the **program steps**.
- **Program Step:** It is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time, which is independent of instance characteristics.
- Example: return(a+b+b*c+d/(a-b*c)) $\rightarrow$ treated as a single Program step only.
- Two ways to determine the number of steps needed by program to solve a particular problem instance.
  - Global Count Variable Method
  - Tabular Method

- count is new global variable with initial values as 0.
- Statements to increment count by appropriate amount are introduced into the program.
- Each time the statement in original program is executed, count is incremented by one.
- This count variable resembles the Program steps, which specify time complexity.
- Example: Time Complexity Calculation using Global Count Variable for Summation of n-numbers.

Time Complexity

- Time Complexity Calculation using Global Count Variable for Summation of n-numbers

Algorithm Sum(a,n)

```
{
sum ← 0;
for i ← 1 to n do
    sum ← sum + a[i];
end for
}
```

Time Complexity

- Time Complexity Calculation using Global Count Variable for Summation of n-numbers

Algorithm Sum(a,n)

```
{
sum ← 0;
count ← count + 1
for i ← 1 to n do
    count ← count + 1
    sum ← sum + a[i];
    count ← count + 1
end for
count ← count + 1
}
```

Time Complexity

- Time Complexity Calculation using Global Count Variable for Summation of n-numbers

---

Algorithm Sum(a,n)

{
$sum \leftarrow 0$;
for $i \leftarrow 1$ to n **do**
    $sum \leftarrow sum + a[i]$;
    $count \leftarrow count + 2$ // repeats n-time
end for
$count \leftarrow count + 2$
}

- So the Time Complexity is T(Sum) = t_{Sum}(n) = **2n+2**

- Algorithm for Matrix Addition $\rightarrow$ Time Complexity $\rightarrow$ Global Count method

Algorithm MatrixAdd(a,b,n,m)

```
{
for i ← 1 to n do
    for j ← 1 to m do
        c[i][j] ← a[i][j] + b[i][j];
    end for
end for
}
```

Time Complexity

- Algorithm for Matrix Addition → Time Complexity → Global Count method

Algorithm MatrixAdd(a,b,n,m)

```
{
for i ← 1 to n do
    count ← count + 1          // i^th for loop
    for j ← 1 to m do
        count ← count + 1          // j^th for loop
        c[i][j] ← a[i][j] + b[i][j];
        count ← count + 1          // for addition logic
    end for
    count ← count + 1          // j^th for loop termination condition
end for
count ← count + 1          // i^th for loop termination condition
}
```

Time Complexity

- Algorithm for Matrix Addition $\rightarrow$ Time Complexity

Algorithm MatrixAdd(a,b,n,m)

$\{$
for $i \leftarrow 1$ to n **do**
    $count \leftarrow count + 2$ //ith loop true and jth loop false cases
    for $j \leftarrow 1$ to m **do**
        $count \leftarrow count + 2$ // jth loop true cases and logic
        $c[i][j] \leftarrow a[i][j] + b[i][j]$;
    end for
end for
$count \leftarrow count + 1$ //ith loop false cases
$\}$

- Time Complexity using Count variable is **2mn+2m+1**

Time Complexity

- Summation using Recursion → Time Complexity → Count variable method

Algorithm RSum(a,n)

```
{
if (n ≤ 0) then
    return 0;
else
    return RSum(a,n-1)+a(n);
end if
}
```

Time Complexity

- Summation using Recursion → Time Complexity → Count variable method

Algorithm RSum(a,n)

```
{
count ← count + 1     //for the IF conditional
if (n ≤ 0) then
    count ← count + 1     //for the return
    return 0;
else
    count ← count + 1 // for the addition, invocation and return
    return RSum(a,n-1)+a(n);
end if
}
```

- Assume $t_{RSum}(n)$ is the runtime of the above Recursive Algorithm.
- if $n = 0$ then $t_{RSum}(0)$ is 2.
- if $n \geq 0$ then count increments by 2 and time taken to execute invocation RSum() from else part.
- Uses Recursive Formula to counting step count for recursive algorithms
- The Recursive Formulae are called as Recurrence Relations.
- They are many ways to solve the recurrence relations.
- One of the method to solve recurrence relations is Substitution Method
- Substitution Method:
  - Repeated Substitutions for each occurrence of the function t_{RSum} on the right side until all the occurrences disappear.

- The Recurrence Formula for the above Algorithm is

$$t_{RSum}(n-1) = \begin{cases} 2 & \text{if} \quad n = 0 \\ 2 + t_{RSum}(n-1) & \text{if} \quad n > 0 \end{cases} \tag{1}$$

$$\begin{aligned} t_{RSum}(n) &= 2 + t_{RSum}(n-1) \\ &= 2 + 2 + t_{RSum}(n-2) \\ &= 4 + t_{RSum}(n-2) \\ &= 4 + 2 + t_{RSum}(n-3) \\ &= 6 + t_{RSum}(n-3) \\ &. \\ &. \\ &= n(2) + t_{RSum}(0) \\ &= 2n + 2, \qquad n \geq 0 \end{aligned} \tag{2}$$

- Tabular Method
  - The second Method to determine step count is Tabular Method.
  - It depends on Steps for Execution(s/e).
  - The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.
  - Total Number of times that s/e is taken place in algorithm.
  - These two quantities gives the Step Counts of the Algorithm

Statement	s/e	Frequency	Total Steps
Algorithm Sum(a,n)	0	-	0
{	0	-	0
s $\leftarrow$ 0;	1	1	1
for i $\leftarrow$ 1 *to* *n* *do*	1	n+1	n+1
s $\leftarrow s + a[i]$;	1	n	n
return s;	1	1	1
}	0	-	0
Total			2n+3

- Tabular Method for Recursive Algorithm.

Statement	s/e	frequency		total steps	
		$n=0$	$n>0$	$n=0$	$n>0$
1 **Algorithm** $\mathsf{RSum}(a,n)$	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 $\mathsf{RSum}(a, n-1) + a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

$$x = t_{\mathsf{RSum}}(n-1)$$

Time Complexity

- Another Approach to estimate Time Complexity : **Operation Count**
    - It used one or more Operations to specify the time complexity of the algorithm
    - It is not considering all the steps used in algorithm
    - The success of this method depends upon the identification of operations, which contributes more to the time complexity.

Algorithm Max(a,n)

```
{
Max ← a[0];
for i ← 1 to n do
    if Max ≤ a[1] then
Max ← a[i]
end for
return i;
```

- Key Operation: Number of Comparisons in made between elements of Array.
- If the size of Array is zero $\rightarrow$ Number of Comparisons are zero.
- If array has only one element $\rightarrow$ It will not enter into the for loop $\rightarrow$ Number of comparisons are zero.
- When n $>$ 1, each iteration of for loop makes one comparison between the elements of array.
- So, the total number of comparisons is maxn-1,0.
- In this method, It is not including the comparison operation of for loop and other operations also.
- Disadvantage: This Method is not considering the entire algorithm. some of the operations are considered for estimating the time complexity.

Space Complexity

- Space Complexity = Fixed part + Variable Part
- $S(P) = c + S_P$(Instance Characteristics)
- Fixed Space Requirement
  - Independent of the characteristics of the inputs and outputs
  - Instruction Space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirement
  - Depend on the instance characteristic
  - number, size, values of inputs and outputs associated with Instance Characteristics
  - recursive Runtime stack space, formal parameters, local variables, return address

- Determine the Space Complexity for the following example:

```
Algorithm abc(a, b, c)
{
    return a + b + b * c + (a + b - c)/(a + b) + 4.0;
}
```

- The problem instance is characterise by a, b and c.
- Assume one word is adequate to store.
- Space needed by above algorithm is independent of the instance characteristics
- $S(abc) = 3 + 0 \Rightarrow S(abc) = 0$

Space Complexity

- Space Complexity calculation Example

Algorithm Sum(a,n)

```
{
sum ← 0;
for i ← 1 to n do
    sum ← sum + a[i];
end for
}
```

- The above algorithm is characterised by n.
- The space need by n is one word.
- The space needed by a is atleast n words.
- S(Sum) $\geq$ (n+3) $\rightarrow$ n for a[], one word for each n, i and s

Space Complexity

- Recursive Algorithm for summation of n numbers:

Algorithm RSum(a,n)

```
if (n ≤ 0) then
    return 0;
else
    return RSum(a,n-1)+a(n);
end if
```

- Here the instance characteristic is n.
- The recursion stack space includes the space for the formal parameters, the local variable and the return address.
- Assume one word used for return address.
- The depth of the recursion is (n+1)
- Recursion Stack Space needed is $\geq 3(n+1)$.