Complexity Analysis of Algorithms

ALGORITHM ANALYSIS

An algorithm requires following 2 resources:

1.**Memory Space:** Space occupied by program code and the associated data structures.

1.**CPU Time:** Time spent by the algorithm to solve the problem.

Thus an Algorithm can be analyzed on two accounts *space* and *time*.

Factors governing execution time of a program:

1. The speed of the computer system.
2. The structure of the program.
3. Quality of the compiler that has compiled the program.
4. The current load on the computer system.

It is thus better we compare the algorithm based on their relative time instead of actual execution time.

Big-Oh Notation:

The **Big-Oh** notation defines that for a large number of input 'n', the growth rate of an algorithm T(n) is of the order of a function 'g' of 'n' as indicated by the following relation:

$$T(n)=O(g(n))$$

The term '*of the order*' means that T(n) is less than a constant multiple of g(n) for $n>=n_{o.}$

Therefore, for a constant 'c' the relation can be defined as:

$$T(n) <= c*g(n) \text{ where } c>0$$

- From the above relation we can say that for a large value of n, the function ‘g’ provides an **upper bound on the growth rate ‘T’.**

- **Order Of Big-Oh Notation:**

$$O(1) < O(\log_k n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(10^n)$$

Determination of Time Complexity for simple algorithms

- Because of the approximations available through Big-Oh , the actual T(n) of an algorithm is not calculated, although T(n) may be determined empirically.

- Big-Oh is usually determined by application of some simple **5 rules**:

RULE #1:

Simple program statements
are assumed to take a
constant amount of time which is

O(1)

i.e...
Not dependent upon n.

RULE #2:

Differences in execution time of simple statements is ignored

RULE #3:

In conditional statements,

the worst case is always used.

RULE #4:

The running time of a sequence of steps has the order of the running time of the largest

(the sum rule)

RULE #4 (example)

- if two sequential steps have times O(n) and O(n^2), then the running time of the sequence is O(n^2).

- If n is large then the running time due to n is insignificant when compared to the running time for the squared process (e.g... if n=1000 then 1000 is not significant when compared to 1000000)

RULE #5:

If two processes are constructed such that second process is repeated a number of times for each n in the first process, then O is equal to the product of the orders of magnitude for both products

(the product rule)

RULE #5 (example)

- For example, a two-dimensional array has one for loop inside another and each internal loop is executed n times for each value of the external loop.

- The O is therefore n*n

Rules for Big-Oh (Summary)

If T = O(c*f(n)) for a constant c, then
T = O(f(n))

If T1 = O(f(n)) and T2 = O(g(n)) then
T1 + T2 = O(max(f(n), g(n))) → **the sum rule**

If T1 = O(f(n)) and T2 = O(g(n)) then
T1 * T2 = O(f(n) * g(n)) → **the product rule**

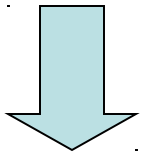
If f is a polynomial of degree k then
$f(n) = O(n^k)$

Code Features

- Assignment : O(1)
- Procedure Entry : O(1)
- Procedure Exit : O(1)
- if A then B else C :
  - time for test A + O(max{B,C})
- loop :
  - sum over all iterations of the time for each iteration.
- Combine using Sum and Product rules.
- Exception: Recursive Algorithms

If f is a polynomial of degree k then **$f(n) = O(n^k)$**

$$f(n) = n^2 + 100n + \log_{10} n + 1000$$



$$f(n) = O(n^2)$$

Show that $3(n+1)^7 + 2n\log n$ is $O(n^7)$

Solution Let us apply rules

- $\log n$ is $O(n)$
- $2n\log n$ is $O(2n^2)$
- $3(n+1)^7$ ⇒ is a polynomial of degree 7, therefore it is $O(n^7)$
- $3(n+1)^7 + 2n\log n$ is $O(n^7 + 2n^2)$

  ↳ is $O(n^7)$

Example showing how to calculate Big-Oh Notation:

Example I: Find time complexity of following algorithm-

Algorithm areaTriangle(base, height, area)
{
Step
1.Read base, height
2.area=0.5*base*height
3.Print area
}

- The algorithm takes **3** steps to complete. Thus

T (n)=3

- To satisfy the relation **T (n)<=c*g (n),** the above relation can be written as

T (n)<=3 * 1

- Comparing the above two relations we get

c=3, g (n)=1

- Therefore, the above relation can be expressed as: **T (n) = O (g (n))**

T (n) = O (1)

- We can say that the time complexity of above algorithm is **O (1)** i .e of order 1

Example II:

Algorithm sum(a[n], sum)
{
Step:
1. sum=0
2.Enter n.
3.for(i=0 to n)
    3.1 Enter a[i]
    3.2 sum = sum + a[i]
4.Print sum
}

- The algorithm takes 2n+3 steps to complete. Thus

T (n)=2n+3

- To satisfy the relation **T (n)<=c*g (n),** the above relation can be written as

T (n)<=2n+3

T (n)<=2n+n

T (n)<=3n

- Comparing the above two relations we get

c=3, g (n)=n

- Therefore, the above relation can be expressed as: **T (n) = O (g (n))**

T (n) = O (n)

- We can say that the time complexity of above algorithm is **O (n)** i .e of order n

Find the time complexity

```
1. if ( x > y )
        x = x + 1
   else
   {
       for ( i = 1 ; i <= n ; i++ )
           x = x + i;
   }

   .
```

Exercise

- $T(n) = 2527$
- $T(n) = 8 * n + 17$
- $T(n) = 5 * n^2 + 6$
- $T(n) = 5 * n^3 + n^2 + 2*n$

To understand these rules, some simple algorithms will be analyzed in their code form.

There are features in the code which allow us to determine O() by applying the rules in the previous notes.

More Examples

- Nested Loops
- Sequential statements
- Conditional statements
- More nested loops

Nested Loops

- Running time of a loop equals running time of code within the loop times the number of iterations
- Nested loops: analyze inside out

```
1 int k=0;
2 for ( int i=0; i<n; i++)
3       for ( int j=0; j<n; j++)
4           k++;
```

O(n)

O(1)

Nested Loops

- Running time of a loop equals running time of code within the loop times the number of iterations
- Nested loops: analyze inside out

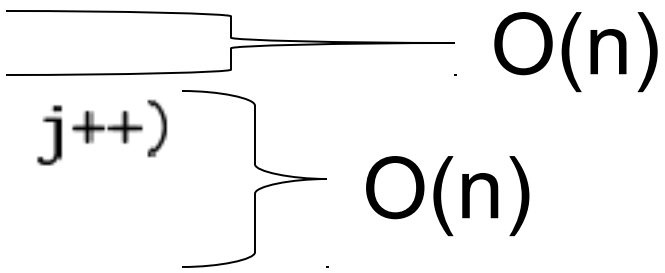
```
1 int k=0;
2 for ( int i=0; i<n; i++)
3       for ( int j=0; j<n; j++)
4           k++;
```

O(1*n) = O(n)

Nested Loops

- Running time of a loop equals running time of code within the loop times the number of iterations
- Nested loops: analyze inside out

```
1 int k=0;
2 for ( int i=0; i<n; i++)
3       for ( int j=0; j<n; j++)
4           k++;
```



Nested Loops

- Running time of a loop equals running time of code within the loop times the number of iterations
- Nested loops: analyze inside out

```
1 int k=0;
2 for ( int i=0; i<n; i++)
3       for ( int j=0; j<n; j++)
4           k++;
```

$O(n*n) = O(n^2)$

Nested Loops

- Running time of a loop equals running time of code within the loop times the number of iterations
- Nested loops: analyze inside out

```
1 int k=0;
2 for ( int i=0; i<n; i++)
3       for ( int j=0; j<n; j++)
4           k++;
```

O(n)

$O(n^2)$

- **Note:** Running time grows with nesting rather than the length of the code

Sequential Statements

- For a sequence S1; S2; : : : Sk of statements, running time is maximum of running times of individual statements

```
1 for ( int i=0; i<n; i++)
2    X[i] = 0;
3 for ( int i=0; i<n; i++)
4      for ( int j=0; j<n; j++)
5          X[i] += i+j;
```

O(n)

$O(n^2)$

Running time is: max(O(n), $O(n^2)$) = $O(n^2)$

Conditional Statements

- The running time of

  if (cond) S1

  else S2

→is running time of *cond* plus the max of running times of *S1* and *S2*

```
int k=0;
for ( int i=0; i<n; i++)
  if even(i) {
     for ( int j=0; j<n; j++)
        k++; }
  else
      k *= 2;
```

O(n)

O(1)

Conditional Statements

- The running time of

  if (cond) S1

  else S2

→is running time of *cond* plus the max of running times of *S1* and *S2*

```
int k=0;
for ( int i=0; i<n; i++)
  if even(i) {
     for ( int j=0; j<n; j++)
        k++; }
  else
      k *= 2;
```

= O(max(n, 1))

= O(n)

More Nested Loops

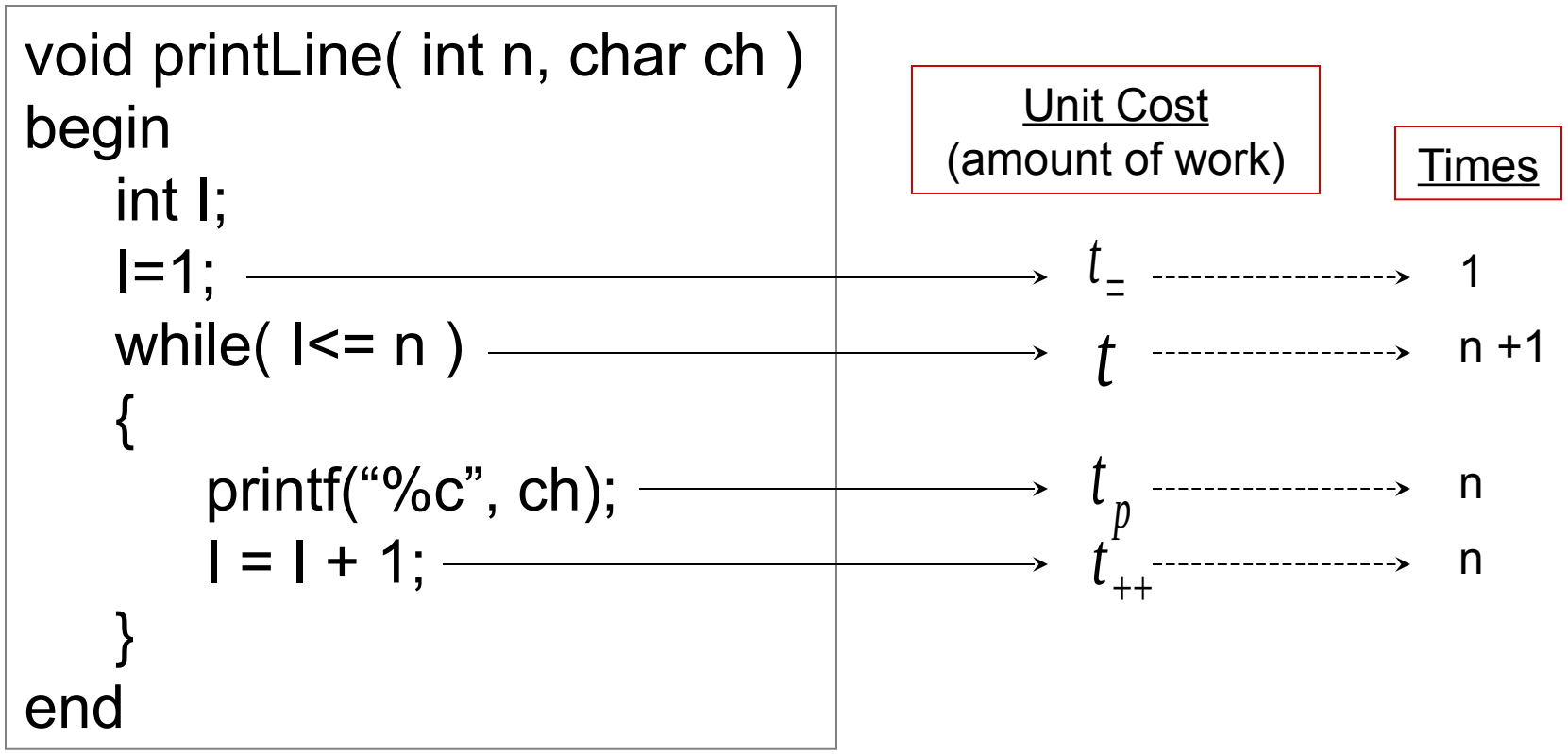
```
1 int k=0;
2 for ( int i=0; i<n; i++)
3       for ( int j=i; j<n; j++)
4           k++;
```

$n-i$

?

$$\sum (n-i) = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = O(n^2)$$

Example : Counting For-Loops



t_{goto} → for the branching structure of *while*

Example : Counting For-Loops

Total time taken:

$$t = t_{=} + (n+1)\,t + nt_{p} + nt_{++} + nt_{goto} \qquad \text{(eq 1)}$$

If $t_{=} = t = t_{++} = t_{goto} = c$

Then (eq 1) simplifies to: $c(3n+2) + nt_{p}$

which is linearly proportional to n → $O(n)$

Example : Bubblesort

```
Void Bubblesort( int[] A ) // A[1…n]
1. begin
2. for i = 1 to n-1 do
3.      for j=1 to n-i do
4.           if A[j] > A[j+1] then
5.                swap A[j] with A[j+1]
6. end
```

Line 1,6: O(1)
Line 4,5: O(1)
Line 3-5: O(n-i)

Line 2-5: $O\left(\sum_{1}^{n-1}(n-i)\right) = O\left(n(n-1)-\sum_{1}^{n-1} i\right) = O\left(n^2\right)$

Basic Asymptotic Efficiency Classes

Class	Name	Comments
1	Constant	Algorithm ignores input (i.e., can't even scan input)
lgn	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)
nlgn	"n-log-n"	Some divide and conquer
n^2	Quadratic	Loop inside loop = "nested loop"
n^3	Cubic	Loop inside nested loop
2^n	Exponential	Algorithm generates all subsets of n-element set
n!	Factorial	Algorithm generates all permutations of n-element set